The Dissertation Committee for François Gérard Barbanson certifies that this is the approved version of the following dissertation:

Active Learning and Compilation of Higher Order Schema Integration Queries

Committee:

Daniel P. Miranker, Supervisor

Kathleen S. Barber

Inderjit S. Dhillon

Raymond J. Mooney

Bruce W. Porter

Active Learning and Compilation of Higher Order Schema Integration Queries

by

François Gérard Barbanson, I.C.M.; M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2005

Active Learning and Compilation of Higher Order Schema Integration Queries

Publication No._____

François Gérard Barbanson, PhD

The University of Texas at Austin, 2005

Supervisor: Daniel Miranker

After nearly 30 years, database integration remains the province of engineers and application developers. In an informal proof, Krishnamurthy, Litwin and Kent [KLK91] demonstrated that only higher order relational languages such as SchemaSQL and SchemaLog [LSS96] are general enough to concisely describe the merging of data from multiple heterogeneous sources. However, those languages are incrementally harder to program than SQL. A modern solution is to provide a GUI language with the same capabilities. However, a Query-by-Example (QBE) [Zloof77] type interface does not allow the unambiguous specification of higher order data integrating queries.

We propose an architecture comprising three layers. In the middle layer, the user expresses the desired federated view through a QBE inspired user interface. Further learning proceeds via a sample selection method asking the user to validate examples of federated records. This interaction ends when the system is satisfied it has converged to the exact view definition the user intends. The bottom layer provides the execution mechanism for higher order data manipulations by compiling higher order relational definitions into first-order SQL programs. The top layer component of the architecture assists the learning algorithm by collecting meta-data and catalog statistics.

Our primary contributions comprise a taxonomy examining trade-offs between complexity and completeness and identifying various classes of higher order relational

data manipulations. The architecture delimits three separate challenges, which must be overcome in order to propose a solution. Our compilation for SchemaSQL proposes novel theoretical complexity guarantees. Type-based vertical partitioning of the meta-data ensures that the result can be properly optimized by existing SQL engines. Sample selection constraints specific to databases require the introduction of a third kind of instance label in the training set of our learner. We derive a new algorithm by modifying Mitchell's version spaces [Mitchell82] in order to handle this new kind of label. We prove that the modified algorithm preserves the original properties of version spaces and avoids the possibility of deadlock. We introduce a sample selection heuristic that converts catalog statistics into a classic inductive bias. Finally, we develop the Sphinx prototype, carry out experiments and demonstrate the system on an application.

Table of Contents

Chapter 1 Introduction

We start by introducing the heterogeneous database integration problem. We note that beyond the traditional distributed query answering formulation of the problem, the need for data conversion, data integration and other multi-database cross-platform data exchange is greater than ever.

We will survey the issues of specifying data integration queries through a learning interface and the various degrees of complexity involved. We catalog our findings in a taxonomy and review related work with that perspective.

We present an architecture for the development of a solution and separate each set of challenges into a different component. We will address each area of this architecture. First, we prove that compiling higher order queries into standard SQL is an effective implementation method with theoretical complexity guarantees. Second, we tackle the problem of an interactive learning approach for data integrating queries. We identify a class of queries from the taxonomy, which is both general enough to represent an interesting solution, and structured enough, to allow us to frame an inductive learning solution. We show how the user can define a search space by dragging and dropping data values to construct a single graphical input example. We present the Version Spaces inductive learning algorithm, which will converge to a solution using a simple question based interaction with the user. Third, we present a heuristic motivated for the sample selection problem, such that the catalog statistics of the input databases can be exploited to derive an information gain estimate.

We implement this solution in a system named Sphinx, and carry out experiments on several case studies. Finally, we demonstrate Sphinx on an application.

1. Historical Perspective

Relational Algebra was first introduced over 30 years ago, establishing the logical foundation for databases [Kuhns67], [Codd70]. A wave of query languages soon followed, including SQL [Chamberlin76] and QUEL [SWK76]. A graphical language called Query-By-Example (QBE) was also conceived at that time [Zloof77]. The goal behind the design of QBE was to have users specify queries in a more natural and less abstract environment than with other query languages. An important claim behind QBE was a study suggesting a much shorter learning period for new users compared to other languages. An example of a QBE query is shown in Figure 1. The query appears as a set of table skeletons populated with symbol placeholders such as the variables "x" and "y" or the command "P". These placeholders appear visually in the same place as the data they represent (for variables) or the data they act upon (for commands).

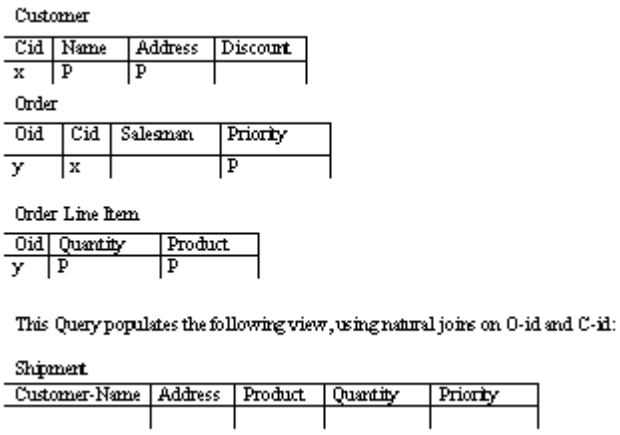


Figure 1- A QBE query

QBE is commonly accepted as the common ancestor of all modern graphical database query interfaces. Figure 2 shows the MS Access database query system. Two algebraic aspects of the query (Join and Projection operators) are described semi-graphically as in QBE. The remaining Selection operators (commonly called filters) are written into the appropriate result column: another QBE concept.

Query models such as QBE, when limited to their intuitive and common sense graphic operations, have less expressive power than SQL or relational algebra. The occasional difficult query can be handled either by reverting to SQL, or by introducing additional non-intuitive graphical manipulations. The tradeoff is that for the vast majority

of queries most users require, the interface is simple enough to handle without the assistance of a database engineer.

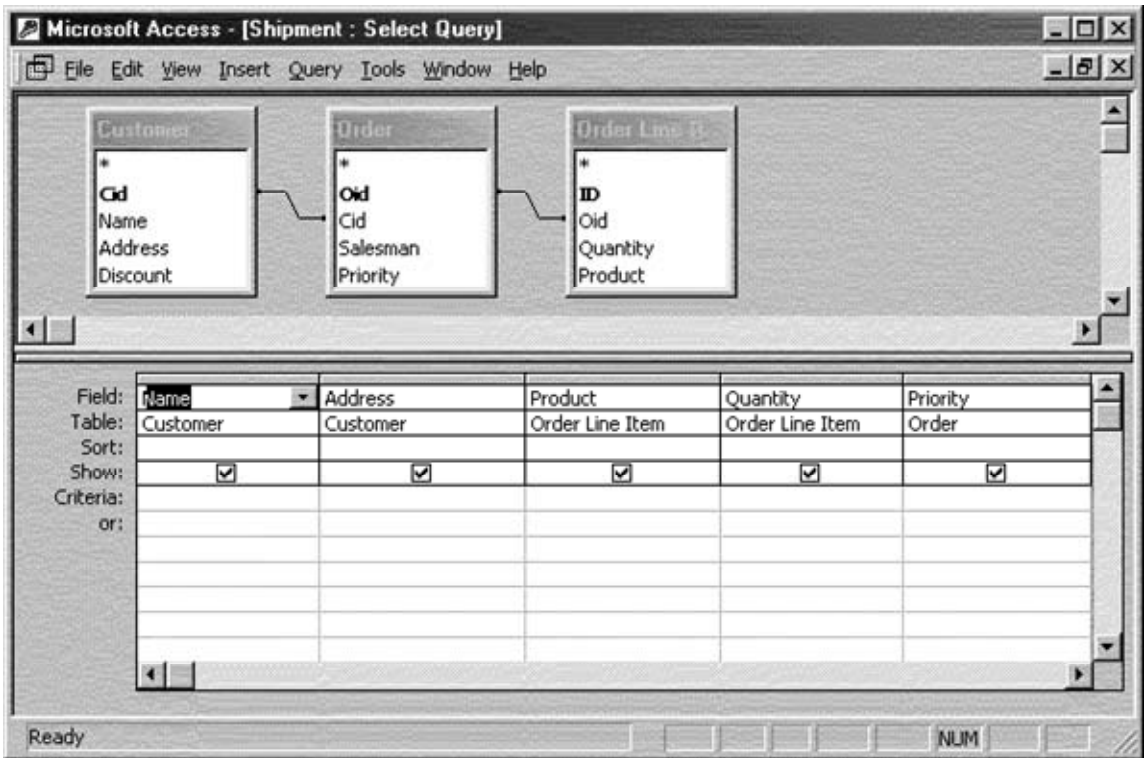


Figure 2 - MS Access Interface

2. Querying Multiple Databases

With the advent of the Internet, nearly every database in the world is accessible from your own computer. This availability is pushing the demand for connectivity standards and querying multiple databases simultaneously has become a reality.

Building a federated system can be approached from many angles. Large proprietary systems traditionally opt for a monolithic approach: data federation involves a large engineering effort to build an ad-hoc one-time solution. Finding answers to this problem has been the focus of research on mediated systems and architectures. The bulk of this research has focused on distributed query answering with redundant,

complementary or heterogeneous sources. However, the nature of the problem is such that in practice new systems are constantly being engineered. This work is the purview of specialized consultants.

Application platforms that integrate internal data storage facilities form another challenge. Such applications, initially confined to web services or information management systems have become commonplace and compete with completely integrated and proprietary vertical solutions. Many applications, especially in the scientific community may allow data import and export as part of a back-end API. Such APIs allow these systems to communicate with specialized software, tools or systems further up or down the information chain. Specialized scripts and ad-hoc import/export utilities handle the conversions between different data standards. Facilitating data connectivity between such heterogeneous applications requires the same data manipulations as interoperating heterogeneous databases. However, some of the constraints are different, and the work is considered off-line: there is no need to document source query capabilities or to build distributed query answering facilities.

3. Why is Building Queries over Multiple Databases Hard ?

Exploiting multiple sources involves data manipulation to move data between native schema content and a federated schema. The specification of a federated view with respect to the tables of a native schema is usually a view defining relational query. The view definition shown below defines a view called “Invoice” as the result of a SQL query (the query produces the same output as the QBE query in Figure 1). The concept of a view itself is similar to the notion of table. The differentiating characteristic is that a view is defined as the result of a query on other views or tables, rather than as the result of insertions.

Create View Invoice as

Select (c.name, c.address, o.priority, i.quantity, i.product)

From Customer c, Order o, Order_Line_Item i

Where (c.cid = o.cid) and (o.oid = i.oid)

In 1991, Krishnamurthy, Litwin and Kent demonstrated through a real life example, that a higher order relational language was required to efficiently federate heterogeneous databases [KLK91]. Krishnamurthy's convincing argument is that a query defining data federating views almost always needs to incorporate variables ranging over schema elements: attribute names, table names, etc.. This requires a logic capable of such manipulations: a second order logic.

Several second order languages have been proposed. One application independent general-purpose language is Schema-SQL [LSS96], which contains the kind of second order elements shown in Figure 4. The syntax is almost identical to SQL, except for the placement of variables where SQL would allow only constants, such as a table name or a column name. In Figure 4, a sample SchemaSQL query uses a variable *s*, as an attribute variable. In several mediator systems that have been proposed [GMP97], [VP97], mediators accept specialized languages (often based on SQL) for the specification of relational transformations, but with additional features to handle meta-specifications, meta-data definitions, and in some cases source query specifications.

The data manipulation queries required to federate heterogeneous sources require more complex thinking and problem solving skills than traditional queries. The general-purpose language to express them, SchemaSQL, is incrementally more powerful than SQL. SchemaSQL uses second order variables to efficiently express queries that would require exponentially larger SQL programs.

These observations lead to the conclusion that using a more expressive language such as SchemaSQL, is a necessary progression to efficiently describe complex data federating queries. However, such a language is not particularly well suited to anyone who is not a specialized data engineer. Further, writing data federating queries in

SchemaSQL, or in any mediator specification language, requires engineers with much more specific training than simply writing ANSI SQL.

Writing a federating query over multiple databases is hard, because it requires using a more expressive and complex language than writing queries over a single schema.

4. Why the Emergence of XML does not Resolve this Problem ?

Document querying languages such as XQuery, XPath and XSLT feature flexible syntactic constructs that allow the specification of document queries in a much more succinct way than SQL. The intent of XQuery is not only to query the contents of a XML document, but also to enable all the possible document transformations foreseeable in the XML domain. Thus it is a complex, powerful and redundant language, and only limited subsets have been implemented to date [XQuery]. Just as ad-hoc solutions and database engineers perform data interchange in the relational world, XML data exchange is performed by scripts and small import/export utilities written by specialists. The problem of data integration and data transformation between heterogeneous XML schemas substitutes itself to the original relational problem.

For data domains and applications where cooperation between independent data managing entities is required, a consensus often builds to adopt one or several XML standards for data interchange. In this cooperative approach each individual data application is responsible for providing import and export facilities using the relevant XML standard(s). The difference with a monolithic approach, is that adopting the common standard shifts the engineering burden from a centralized federating operator to the concerted efforts of several entities.

Ideally, when a single XML standard exists for a given domain, subsequent applications may be designed with data models that are based on the existing standard. However, as in the relational world, the existence of legacy data models and the specific needs of individual applications guarantee the continued need for XML data exchange solutions. The fundamental underlying problem remains unresolved.

5. Why is Building a Graphical Interface for Federating Databases Hard ?

Figure 3 shows how a row of output would be built in a typical interface. The user drags elements of source tables together. In this specific example, a column name is being manipulated by the user and is present in the output. The name of a column is a schema element, and in this case can also be part of the required output data. With such an interface the user incrementally defines each federated view separately by dragging and dropping elements from data sources.

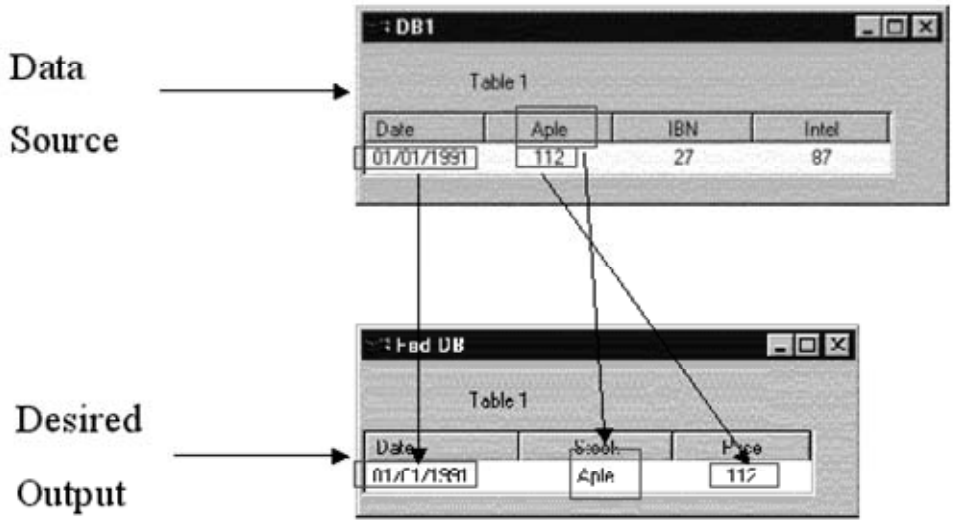


Figure 3 - User Input

The kind of graphic interface shown in Figure 3 would represent a natural way for a user familiar with the data to specify a query defined over multiple data sources (or component databases). However, the input shown in these diagrams does not give a complete specification of a second order query. There is ambiguity as to how to interpret the drag and drop operations into a structured query. Some of the missing elements can

be interpolated, others will have to be extrapolated.

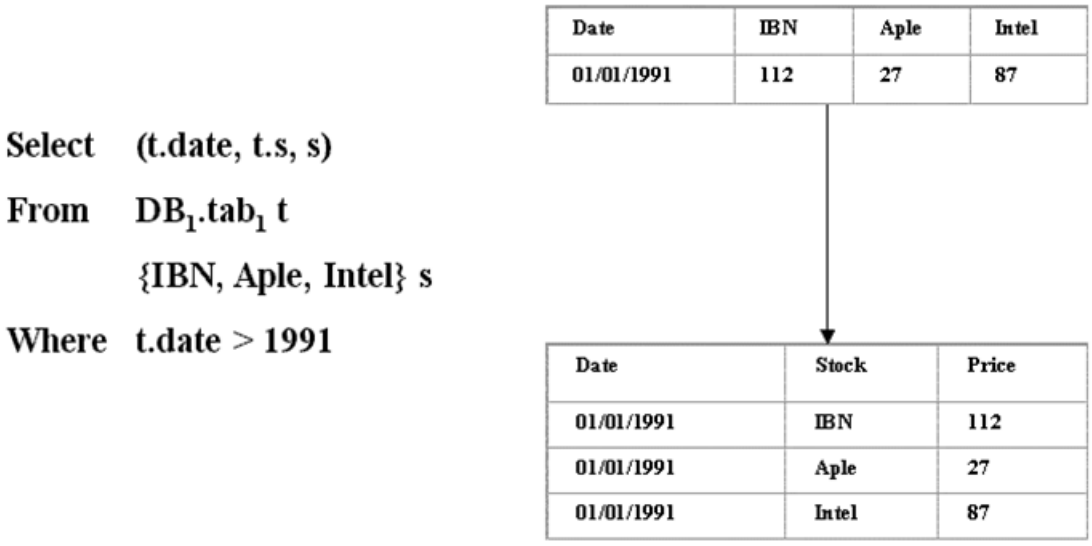


Figure 4 - Data Federation Example

Figure 5 presents two valid expressions for the kind of transformations that are consistent with the graphic input in Figure 3. The first expression is the only pure SQL sentence consistent with the user interaction. SchemaSQL however is a lot more expressive, and the second expression is a blueprint for the many SchemaSQL sentences, that are consistent with the same user interaction. Due to the heightened complexity of data federating transformations, it will be up to the learning system to make choices between competing interpretations when ambiguity exists. The actual query sought by the

user, and its result, are shown in Figure 4.

1st order structure:

```
Select    (t.date, t.Aple, Aple)
From      DB1.table1          t
Where     [...]
```

2nd order structure:

```
Select    (x.y, x.z, z)
From      v.w       x
          w         f1(v)
          ...
Where     f2[u,v,w,x,y,z]
```

Figure 5 - Possible Queries

Building a graphical interface for federating databases is hard because of the ambiguity resulting from a high-level point and click user interaction.

6. Goal

Data federating queries can be expressed in a second order, augmented language of SQL. However, the task of federating databases is complicated by the need to understand and bridge the discrepancies between many large-scale schema and their internal dependencies and constraints. The volume of attributes, views, rules and other schema elements that must be handled to form a complete specification also lengthens that task.

Conversely the increased availability of data sources at the fingertip of users has raised their expectations to casually federate data from multiple sources. Controlling the complexity of this task by providing a set of tools rather than completely custom-engineered solutions has a chance to meet some of those expectations. A reasonable interface paradigm to facilitate this task should allow data definition and manipulation operations in a graphical and intuitive manner. The challenges that we need to resolve in order to build such an interface are twofold.

The first fundamental issue is that integrating data from heterogeneous schema requires more expressive paradigms than SQL, and our system will need to bridge this gap by handling higher order concepts such meta-data variables in SchemaSQL.

The second fundamental difficulty lies in transitioning imperative programming towards a more natural language. Natural language such as English, are intuitive for people but poorly suited to the specification of precise abstract or mathematical concepts. English is ambiguous both syntactically and semantically. Similarly, easy to use and easy to understand point-and-click graphic interfaces, have a limited range of intuitive operations and there is inherent ambiguity in trying to capture complex specifications through them.

Thesis: The goal of this dissertation is to reduce the complexities of querying heterogeneous databases. To do so we will solve the problem of bridging the complexity gap between heterogeneous schema integration specifications and SQL relational expressions in a computationally efficient fashion, and we will solve the latent ambiguity of GUI input by proposing a GUI based active learner.

Query by Example allowed first order database queries to be adapted to modern graphic user interfaces. The goal here is to do propose a similar conceptual step forward for second order queries over multiple databases. Further, we propose to formulate our work by addressing and resolving separately the expressiveness and the ambiguity gap.

7. Taxonomy

We will proceed by outlining taxonomy for the purpose of identifying different classes of solutions and their respective trade-offs. The taxonomy examines characteristic features of the problem and will guide our choices in the implementation of a solution.

We measure the problem of learning federating queries from the perspective of three different parameters:

- Theoretical abstraction and difficulty of user interaction. We regroup in this category any element that stands as an obstacle to user interaction. Difficulties will fall under two main categories: requiring the user to learn complex concepts such as a query model or language (like SQL) and requiring the user to perform repetitive operations through the interface.
- Completeness. By allowing the specification of various relational operators, the interface may handle different classes of data manipulations, with some more complete than others.
- Learning complexity of the problem. This is determined by the size of the search space and the complexity of the learning algorithm. Inasmuch as we seek to establish the unambiguous specification of a result through the interface, the prime parameter should be the size of the search space.

The object of the taxonomy is to establish in terms of the third parameter, learning complexity, the cost of providing better interfaces as measured by the first two parameters: quality of user interaction and completeness.

Finally, the ideal interface would allow users to handle elements that are easy to specify with a GUI but are difficult to automate, and trust an inductive learner to extract the kind of knowledge that is difficult to specify over a GUI.

8. Architecture

We will outline an architecture dividing the overall problem into several components that must be addressed in order to reach our goal. In particular we propose that any path to a database integration solution must at the very least deal with three issues:

- Learn about the existing databases
- Learn the federating queries to integrate those source databases
- Execute the federating queries

These three components form a minimal architecture for a database federation interface concept. Each of these components is also at the core of the database integration problem, and is required for any database federation solution.

Learning about the existing databases would be an absolute necessity in any database system. In a relational context, SQL or SchemaSQL programs must include a data definition component, which describes the type and the content of existing tables and of other relational objects. In an interface, this knowledge is first put to use to visualize the database, including tables and dependencies. Second, this knowledge will be put to use to help disambiguate user operations by taking into account features of the data such as join dependencies.

Learning the federating queries to integrate database sources corresponds to learning the data manipulation component of a relational program. Such programs are always written on an individual view definition basis. And more generally, in any system, the federated database is specified one view at a time. Thus the real problem we face is the query discovery problem for view definition.

The execution engine is a necessary component for any data integration solution. The computation of view content as the output of definition queries is called the view materialization process. Any system allowing the visualization and verification of result data by the user requires a way to materialize views. Although we do not plan to let the

user wrestle with the verification problem unaided, we must be able to compute output from various data federating queries if we hope to be able to disambiguate them.

9. SchemaSQL Execution Engine

An efficient execution mechanism for SchemaSQL is necessary to make higher order relational manipulations part of a solution. Execution mechanism for higher order relational languages previously adopted either of two problematic approaches: an inefficient compilation mechanism, or a native execution engine combining higher order and traditional relational operators. The first approach runs the risk of failing in certain cases due to combinatorial explosion when view definitions combine several higher order variables. The second approach simply lacks portability, which is an essential requirement in providing a solution for heterogeneous database systems. An efficient compilation mechanism of SchemaSQL into first-order SQL, presents the advantage of being portable to virtually any database with an SQL interface.

We define such a compilation. Higher order SchemaSQL variables are integrated into first order mechanisms by modeling them as explicit or implicit joins. The compiled program is the aggregation of two parts:

1. Meta-data instantiated in vertically separated type-based views (i.e. intermediate views).
2. The second order program converted into a first order program of the same size by integrating higher order elements via explicit joins with the intermediate views.

The size of a compiled program is strictly linear in terms of the sum of the database catalogs and the original program. The resulting complexity is also linear: higher operators are converted into the *same number* of first order join operators.

10. Learning Phase

We plan to integrate an inductive learning algorithm in our interface. This will provide two advantages: learning with examples taken from the databases and learning a solution from a search space of candidates. Because we do not have a classic training set, in order to completely verify the correct solution the system must take the initiative of constructing examples and submitting them to the user if appropriate.

The first step is to identify a syntactic decomposition of higher order relational queries, creating a correspondence between higher order relational data manipulations and a set of classifiers. We build on the taxonomy to define a search space, by introducing a number of simplifying assumptions. We define the notion of a training instance, which can be labeled by a user and lead to the inductive learning of a classifier.

The second step is to integrate sample selection into this inductive learning. Because we restrict the interface to intuitive but ambiguous operations, we count on an interactive question/answer interaction to lift ambiguity by submitting new examples to the user. This active learning process will continue until the algorithm converges to at most one solution consistent with the user's input on those examples. While we observe that Mitchell's version spaces adequately models this systematic candidate elimination process, adding sample selection and active learning requires adapting version spaces to handle a third kind of instance label in training sets. This new label is designated "missing" and represents the discovery that certain potential example instances cannot be found in the existing databases and cannot be taken into account. Simply ignoring the new label by excluding those instances from the training set introduces the possibility of deadlock. We define a new candidate elimination rule for version spaces allowing us to preserve the original properties of version spaces and avoid deadlock. This rule lets us map some missing labels to negative labels under a specific set of constraints on the training set. We prove that, this modified version spaces correctly keeps track of consistent hypotheses preserving the original properties of Version Spaces. Further, in our sample selection framework, this algorithm is guaranteed to avoid deadlock.

The third challenge is to introduce bias into our search space of federating view definitions. Because of the nature of bias-free learning, the version spaces approach is exponential. We introduce bias based on our domain-related knowledge of federating queries. Exploiting database catalog statistics quantifies an information gain measure that will guide us in the sample selection process. Bias lets us select examples maximizing returns, and brings interaction within reasonable bounds. Our approach will prove experimentally consistent with the original observation that sample selection can provide substantial gains over randomized training sets [CAL92].

11. Prototype

We build the Sphinx prototype, and measure its success in learning federating queries on a real world query discovery problems. We do so by reporting the number of interactive steps necessary for convergence. At each step, the user must examine the example placed before him and render a correct decision as to whether this example represents a federating instance. The number of steps necessary for each query discovery problem represents the burden placed on the user to establish an unambiguous federating view definition.

We demonstrate our system on real world applications. Empirical observation allows us to draw conclusions highlighting both some advantages of the approach (validation of our chosen heuristic bias) as well as some limitations (lack of aggregation operators).

12. Synopsis

Problem Summary:

- Heterogeneous database integration encompasses a wide range of applications from mediated query answering to cross-platform data transfer
- Data federation represents a complex, higher order, data manipulation problem

- User interfaces present a limited set of intuitive operations for the user
- The resulting expressiveness and ambiguity gap must be bridged in order to effect better interface concepts

We propose to contribute to the following areas:

- Database integration and interface taxonomy
- Component-based solution architecture
- Efficient and portable higher order SchemaSQL execution mechanism
- Syntactic framing of higher order SchemaSQL as a classifier search space
- Inductive and Active Learning Framework
- A tri-label version spaces candidate elimination algorithm
- Sample selection heuristics exploiting database catalog
- Demonstration of the Sphinx prototype on an real world applications

Chapter 2 Heterogeneous Databases Integration: Taxonomy and Related Work

This chapter examines the different levels of heterogeneous database integration. We start by presenting a taxonomy from different angles. The ultimate purpose of this taxonomy is to clarify the distinct challenges for a schema integration interface. Current graphic user interfaces and experimental systems also take their place in this taxonomy. We will recapitulate existing and related work, and briefly introduce existing tools and languages for specifying database integration.

1. Taxonomy of View Defining Queries for a Schema Integration Interface

We seek to identify classes of relational transformations for specifying database integration. We look at this problem from the point of view of a learning system trying to discover the federating queries. Because of the complexity of this learning problem, the taxonomy does not have a tree shaped structure. Instead, we need to consider four different aspects of this learning problem. First, we consider the issue of framing the learning problem in terms of target concept and of positive and negative examples. Then, we survey the characteristics of a higher order relational mapping, in terms of syntactic features and in terms of algebraic operators. The features in this classification are relevant to the complexity of the learning problem. The addition of more or less complex syntactic or algebraic features will grow the search space for the learning algorithm accordingly. Next, we consider the learning framework by looking at the type of Data Sets a user could provide for the system to draw upon. The last criteria for the taxonomy is the

complexity of user operations required as an input to the learning system. The degree of required user sophistication is indicative of the target audience for the interface. A couple of charts of the expected interactions between these various aspects of the learning problem are also drawn.

The focus of our research, our goals and simplifying assumptions should be considered in light of the overall complexity of the learning problem emphasized by this wider taxonomy.

1.1. Relational Queries as Classifiers

The goal for building a learning system is to assist an interface in allowing the user to specify a higher order relational mapping. In the broadest interpretation the interface would seek to learn a class of mappings corresponding to any query which can be expressed in a general purpose language for database integration, such as SchemaSQL [LSS96].

Thus it makes sense to consider that the goal of the interface and learning system is to identify a target concept as a higher order relational data manipulation or mapping. But given a set of input databases the target concept can also be modeled as the materialized view it defines rather than as a relational mapping. In that alternate approach the target concept is simply the federated view output by the data manipulation. Computationally the two are not exactly equivalent, since there are more possible view definitions than output views. The concept of relational data manipulation is richer and more detailed than the concept of relational tables or materialized views. Two mappings can produce the same output, but the opposite is not true.

However despite this lack of precision, representing the target concept as a materialized output view has several advantages: positive and negative examples for that concept are a simpler concept for the user. In that case positive examples are simply rows that belong to the target concept, and negative examples are rows that do not belong. We can refer to such examples as *weak examples*.

If on the other hand the target concept is a relational mapping then a complete example instance is formed by both a set of input databases and an output table. In the absence of Group-by operators, relational difference and row ordering in the output view, the target relational transformation is monotonic. For such monotonic mappings, it is sufficient for an example instance to comprise simply of a set of input variables data or meta-data assignments and a corresponding output row for that variable assignment. We can refer to those examples as *strong examples*. One such example is the Graphic Input example shown in Figure 3. This taxonomy will give formal definitions of both kinds of examples.

1.2. Syntactic Complexity

The syntactic structure of SchemaSQL is similar to SQL. Classes of relational transformations are defined in this section by their syntactic structure. To simplify, the general syntax of a SchemaSQL query is the following:

<Query> = Union (Select (<var>
| <var> . <var>
| <var> . <constant>
*| <Vexp>)**
*From (<RANGE> <var>)**
*Where ((∧ / ∨) <Vfor>)**
*)**

<var>	*Variable (1st or 2nd order)*
<RANGE>	*Range Expression (including embedded <Query>)*
<Vexp>	*Expression (including embedded <Query>)*
<Vfor>	*<Vexp><op><Vexp>*
<op>	*= | < | > | ≠*

We start a classification at the top by describing the class of queries with the most expressive syntactic constructs. We progressively restrict this general form to yield more limited classes of queries as we narrow the allowed range of syntactic constructions. We introduce these restrictions in a hierarchical order from the outer to inner syntactic structure.

- Clause Level

The atomic unit of a query or <*Query*> expression is the Select/From/Where (S/F/W) construction shown above. Larger queries can be built by connecting several of these units with relational connectives, or by nesting them inside <*RANGE*> or <*Vexp*> sub-expressions. Several relational connectives exist: such as clause union, difference and intersection, but in practice the main construction is the Union. Relational difference and intersection can be expressed by embedding.

  - Clause Union

In SQL, disjunction is often expressed as a union between two or more clauses. Below is an example of a union between two simple S/F/W clauses.

Example:

(Select (t.date, t.s, s)

From {IBM, Apple, Intel} s

Euter.Table t

*Where *)*

Union (Select (t.date, t.price, s)

From {IBM, Apple, Intel} s

Chwab.s t

*Where *)*

  - Embedded Clauses

A query expression may contain several embedded queries, either in the From line or in the Where line. The first form of embedding can be used to express relational composition between two queries. By embedding, the result of one query serves as the output of the other. This example shows how a sub-query is embedded in the From line of a larger query.

Example:

Select (t.date, t.s, s)

From (Select u.company

From Ource.Table u

*Where *) s*

Euter t

*Where **

The second form of embedding can be used to express either relational intersection or difference. This second example shows how to subtract the result of one query from another by embedding in the Where line.

Example:

Select (t.date, t.s, s)

From {IBM, Apple, Intel} s

Euter.Table t

Where s not in (Select u.stock

From Ource.Table u

*Where *)*

- Single S/F/W (Select/From/Where) or PSJ (Project-Select-Join) Clause (or none of the above)

- Term Level
    - Unit and Domain Conversion Terms

Domain mismatch and measurement unit conversion problems are the other kinds of heterogeneity that arise from Heterogeneous Databases [Kent91]. Consider the following instance of this problem. A list of job titles corresponding to nomenclature appears in the columns of one database table.

SysAdm	SoftwareEngineer	MarketingStaff	ResearchStaff	ProjectDirector

Another database uses a different nomenclature, whose titles adorn the columns of a similar table.

System Engineer	Development Engineer	Consultant	Research Scientist	Program Manager

Given appropriate data correspondences and mappings, this kind of problem can be solved. However acquiring the knowledge necessary for a solution in an automated fashion is a complex learning problem in its own right, and of a very different nature.

Provided that a one to one conversion function JobTitle exists, a solution to this problem would use the following kind of syntactic operator:

Select (person.Name, person.Salary, JobTitle(s))

From {SysAdm, SoftwareEngineer, MarketingStaff, ResearchStaff} s

s person

More complex mappings may be introduced allowing for many-to-one, one-to-many and many-to-many conversion [DDH01], [DLD04].

- Aggregation Terms

Standard aggregation operators, include AVG(), SUM() and other arithmetic functions. Once implemented they can be used in a SQL or SchemaSQL clause to form

an expression. Below is an example query returning the average stock price in the Ource database:

Example:

Select t.stock, AVG(t.price)

From Ource.Table t

- Sort and Group-by Terms

Group-by directive allow the clustering of output tables according to any attribute, which is of an ordered type. Typically such directives are implemented in current graphic user interfaces by checking a box in the appropriate columns.

Example:

Select t.stock, t.price

From Ource.Table t

Where t.date="01/01/2001"

Sort-by t.price

Group-by t.stock

- None of the above

If none of the above syntactic constructions are allowed to appear, relational expressions can only be formed using simple arithmetic operators. Expressions may contain either data or meta-data constants, variables, or composite "dot" constructions such as *row.column*. The number of resulting expressions is limited and the complexity of the learning problem reduced.

- Variable Level
  - Dynamic Schema

In SchemaSQL, it is possible to define views with a schema that is not in normal form. A view may be defined by a query containing an iteration operator in the Select

line. The iteration operator creates a new column for every value of a variable. This construction is used in the example below to map the schema of the database Ource to the schema of database Euter (see Figure 7).

Example:

Select (t.date, [s : t.price])*

From Stocks s

Chwab.s t

- Hidden Variables

A variable that appears in the Where line of a query but does not appear directly or indirectly in the Select line is a *hidden variable.* A variable appears directly if it is part of an expression in the Select line. A variable appears indirectly if it forms the range of another variable somewhere in the From line and that other variable also appears in the Select line. Hidden variables pose a special challenge, because they are not directly implied by the output view and the projection operators. They have to be invented or suggested at the time of the creation of a search space, and there are an infinite number of possibilities for invention. The presence of hidden variables cannot easily be ruled out for any given query since they can always be used to form elaborate filtering predicates or complex join paths. The example below illustrates a query with hidden variable *u*.

Example:

Select (t.name, t.price, v.stock)

From Table1 t

Table2 u

Table3 v

Where t.stock = u.stock

and u.name = v.name

- Self-Join Variables

Two variables drawn from the same range form a Self-Join. The example below shows a self-join between the tables of the Chwab database.

Example:

Select *(t.date, t.price, u.price, s)*

From *{IBM, Apple, Intel} s*

Chwab.s *t*

Chwab.s *u*

Where *t.price < u.price*

- None of the above

- Boolean Level

At this level of the taxonomy we differentiate mappings by the expressions which can appear in the Where line of the Select/From/Where clause.

- Arbitrary DNF

In the most general case the Where line is an arbitrary expression formed by predicate formulas that are separated by the Boolean connectives $\vee$, $\wedge$ and $\neg$. This is equivalent to arbitrary DNF and from a learning point of view there are no known PAC learners for arbitrary DNF. Allowing disjunction and learning with strictly no bias will lead to learning the disjunction of all positive examples. In the version spaces case this will always be the most specific boundary set.

Regardless of those difficulties, even if learning is possible: learning an exact DNF concept is extremely specific. Omitting possible duplicated rows, the set of possible DNF queries is at least as specific as the power set of database rows since for any set of rows a query can be written which selects that exact set.

- k-DNF

Learning arbitrary DNF, and monotone DNF (no negation) can be quite daunting as well as undesirable from our perspective. A k-term DNF expression is the disjunction of k purely conjunctive terms and might form a better basis for a search space.

From our point of view, we can consider k queries separated by k *Union* operators, such that each query is formed by a purely conjunctive composition of predicates in Where line.

In a database context, limiting SQL to disjunctive queries separated by a limited number of *Union* operators (1 or 2) would probably not excessively limit users. Following Occam's razor principle we conjecture that the majority of relational queries have few or no *Union* operators. We offer as evidence the lack of disjunctive in several major database graphical query interfaces (such as MS Access). Besides this presumed bias towards queries with few or no *Union* operators, the built-in INSERT operation, allows a user to aggregate the result of several materialized views without using *Union*.

Example:

[Select (t.date, t.price, s)

From {IBM, Apple, Intel} s

Chwab.s t

Where (t.price < 100)]

Union

[Select (t.date, t.price, s)

From {IBM, Apple, Intel} s

Chwab.s t

Where (t.price > 150)]

- k-CNF

One could also k-term CNF queries as a concept language to express disjunctive queries. In the relational terms, the cost of the *Union* operator is high, whereas conjunction lends itself to efficient optimization. Below are two distinctive disjunctive queries:

Disjunctive Query 1.

Select (t.date, t.price, s)

From {IBM, Apple, Intel} s

Chwab.s t

Where ((t.price < 100) or (t.price > 150))

And ((t.date = 01/01/01) or (t.date = 01/02/01))

Disjunctive Query 2.

Select (t.date, t.price, s)

From {IBM, Apple, Intel} s

Chwab.s t

Where (t.price < 100) or (t.date.year = 2001) or (s <> IBM)

Disjunctive Query 1 uses disjunction to allow selection predicates for scalar variables over non-connected ranges (i.e. union of intervals). Given the proper set of indexes, Disjunctive Query 1 should optimize quite well. Disjunctive Query 2 uses a more general form of disjunction between unrelated predicates and must execute as virtually three separate queries optimized independently. Disjunctive Query 2 incorporates disjunction in the Where line and can be rewritten in DNF, and split using relational *Union*. Such rewriting reflects the status of disjunction as rare and costly. On the other hand, predicates such as *((t.price < 100) or (t.price > 150))* can be seen as a single block expressing a range selection criteria for *t.price*. Range selection predicates treated in that fashion can be efficiently optimized by a query planner. Thus if one considers *((t.price < 100) or (t.price > 150))* and *((t.date = 01/01/01) or (t.date =*

01/02/01)) as a single block each, Disjunctive Query 1 can be viewed as a conjunctive query. Versions spaces algorithms have been shown to handle attributes ranging over continuous scalar intervals [Sebag96].

- Negation

DNF without negation is called monotone DNF because without negation there is no way to retract a fact. Regardless of whether DNF or k-DNF is used, allowing or disallowing negation changes the language. Introducing negation in a language with equality predicates is equivalent to introducing inequality predicates with an attending increase in expressivity. On the other, range selection and arithmetic comparison predicates are unchanged by negation, and represent closed sets with respect to negation.

Example:

Select (t.date, t.price, s)

From {IBM, Apple, Intel} s

Chwab.s t

Where ¬(t.price > 100) and ¬(t.price = 0)

is equivalent to

Select (t.date, t.price, s)

From {IBM, Apple, Intel} s

Chwab.s t

Where (t.price ≤ 100) and (t.price ≠ 0)

- Predicate Level
  - Arbitrary Selection Predicates

Any predicate involving a variable over any kind of formula such as *x = concat(y, z)* or *t.price < AVG(t. price)*.

- Comparison Predicates

An arithmetic predicate can be any arbitrary selection predicate such as *t.price > 100* or *s≠IBM* involving the equality, inequality and comparison operators over ordered or partially ordered domains. Learning comparison predicates over numerical domains is possible with some forms of version spaces algorithm [Sebag96], [NH93].

- Equality predicates

This is the most restrictive case, where all the simplifying assumptions are in force (i.e. none of the above syntactic features can appear). We call that case $S_{0.0}$. An equality predicate is formed by using the equality sign such as in *t.price = 100* or *equals(stock, "IBM")*. This is the only available predicate for nominal valued attribute languages. In the relation database context, tree-structured attribute languages do not occur as there are no hierarchies of values in data type domains.

1.3. Algebraic Complexity

Relational data manipulations can be expressed using either algebra or calculus. SchemaSQL has its algebra, a higher order extension of relational algebra. In this section, we define classes of relational queries by their algebraic formulation using operators and sets. Algebraic and calculus formulations of the higher order query from Figure 4 are shown below. This algebraic classification is almost identical to the syntactic approach, and for most categories, it is unnecessary to repeat the definition of their syntactic counterparts.

- Set Operators
  - Union
  - Difference
  - Intersection
  - None of the above
- Join Operators

    - Hidden Joins
    - Self-Joins
- Other Relational Operators
    - Unit and Domain Conversion
    - Aggregation Operators
    - Group-by Operator
    - None of the Above
- Operable Sets
    - Arbitrary sets of meta-data elements

Catalog elements are enumerated into meta-data types. Those types are catalog meta-data domains and can be used by SchemaSQL to form higher order algebraic expressions.

    - One Meta-Data domain per meta-data element

The elaboration of a default higher order search space from an initial mapping example will require the hypothesis that meta-data domains are disjoint. This implies that catalog elements, such as attributes or tables belong to exactly one meta-data domain, or variable range. This simplifying assumption for an element x is noted as 1MD(‘x’), and implies that a default value for 1MD(‘x’) be assigned if x is not part of a pre-defined meta-data domain. The assumption 1MD(‘x’) does not preclude a catalog element x, from belonging to one or several tables, but it precludes x from appearing in any other meta-data enumeration.

    - Tables only

In a first order algebra, the only operable sets are the tables of the database. In a higher algebra, operable sets also include typed enumerations of catalog elements.

- Boolean Operators

- Negation
- Disjunctions and Conjunctions
- Conjunctions only
- Arbitrary Predicates
- Equality Predicates only

These are not algebra operators. However predicates are allowed to appear inside algebra selection operators.

1.4. Learning Complexity

For consistent learners, the size of the hypothesis space determines an upper bound for the number of examples necessary for PAC learning. We would like to estimate the size of the hypothesis space for each class of query to evaluate both learning complexity and expressive power.

Let $S_{0,0}$ be the minimal search space, applying all the simplifying assumptions that were introduced earlier: single S/F/W clause, no term level or variable level features, 1-DNF and equality predicates as the sole predicate level feature The size of the search space, is the number of legal, single clause S/F/W queries, which are unique to a permutation of the variable names. The number of potential selection predicates for a given variable depends on the domain size, and will be infinite for real-valued attributes.

However after one positive example and data mapping is given by the user, we will prove that the resulting search space $S_{0,1}$ becomes finite (See Graphic Input Theorem). The equality predicate over nominal-valued attribute assumption reduces the number of legal predicates to one per column in the source databases. Further all considerations about the number of possible projection operators are also removed after initial user input.

$$\#of\ conjunctive\ sentences\ (1\text{-}DNF) = 2^{(\#\ of\ predicates)}$$

$$||S_{0,1}|| \leq (\#of\ conjunctive\ sentences) = 2^{k},$$

where k is the number of columns in the source database

Let $S_{1,0}$ be the search space $S_{0,0}$ expanded by allowing negation. For $S_{1,1}$ we do not have a simple result in terms of the cardinality of the search space. A positive example successfully excludes all potential equality predicates but for one per variable, however potential inequality predicates cannot be excluded from the search space so easily.

Let $S_{2,0}$ be the search space $S_{0,0}$ expanded by allowing both disjunction and negation. By considering that the number of DNF sentences is equal to the number of truth tables one can form, a trivial upper bound with respect to the number of predicates. However, the number of predicates after an initial user given positive example is no longer finite since disjunction, unlike conjunction does not rule out mutually exclusive predicates.

$$\#\ of\ truth\ tables = 2\hat{}(2^{(\#\ of\ predicates)})$$

The search spaces $S_{1,0}$ and $S_{2,0}$ are no longer finite. However, most of the target queries in those search spaces are equivalent: on a given set of source databases they will return the same result. The number of equivalence classes for $S_{2,0}$ defined for a given state of the source databases is also an interesting measure. Without self-joins, in a worst case scenario we have a maximal cartesian product defined by including every table in the source databases.

Discounting possible projection operators, the number of possible queries is the powerset of that cartesian product. In the worst case scenario, for completely disjoint attribute domains the number of projection operators is a multiplicative factor on the number of possible views, and is equal to the power set of the source database columns minus 1.

$$||cartesian\ product|| \leq ||maximum\ table\ size||^{(\#of\ tables)}$$

$$||possible\ views|| \leq 2^{||cartesian\ product||+\ k\ -1}$$

By either measure learning target queries that allow a DNF expression in the Where line, is a daunting task, regardless of the PAC-learnability status of DNF. With

DNF and version spaces the most specific boundary set is the set of all positive examples. Convergence even if tractable will yield a disjunctive list equal to the extent of all legal target views: a result of dubious applicability. However disjunction undeniably has a place in relational queries: it is only bias-free learning of DNF which is problematic.

1.5. Data Sets

For some classes that are high on the complexity scale, no amount of weak or strong examples are sufficient to learn a query satisfactorily without additional and richer data sets. For simple target concepts, additional data may yield order of magnitude improvements in the number of necessary examples. It may also serve to disambiguate orthogonal learning issues, enlarging the set of features handled by the system. Here is a proposed list of input types for a federating query learning system.

- L_4: Domain and Unit Conversion

One to one mappings allow conversion from one domain to another. For example consider the input of a dictionary function Φ:{SysAdm, SoftwareEngineer, MarketingStaff, ResearchStaff, ProjectDirector}$\rightarrow$ {System Engineer, Development Engineer, Consultant, Research Scientist, Program Manager}. The specification of this input would immediately allow the system to resolve conversion issues between two heterogeneous domains.

A global mapping such as Φ is a collection of atomic correspondences between pairs of elements. While the specification of each pair is easy and presents a small simple operation for any user, the specification of the global mapping is more problematic. Capturing this kind of input becomes a repetitive and arduous task because of the quantity of small operations required.

More complex, many to many mappings may define how several attributes are mapped onto a set of several other attributes. This makes conversion mappings that much more labor intensive for a user to specify, as well as potentially abstract and complex [DLD04].

- L_3: Aggregation and Group-by Operators

Hinting: This data set could simply involve the user telling the system, which operators are present in the target concept. The system then has to consider the number of expressions that can be built using that operator. Matching output examples with candidate applications of the operator over all available data can form a search space. To give such hints, the user does not need to know any abstract language, but under the concept of average, or sum over data elements.

Specifying: A more complete specification can be given, where the user not only shows which relational operators are present in the target concept, but also builds the necessary expressions for inclusion in the target concept. It is excluded that such specification could be effected with an intuitive graphic manipulation. This would at the very least require the user to master an abstract specification language (such as a spreadsheet formula builder) and perhaps even require knowledge of SQL.

- L_2: Meta-Data Ranges

This kind of data set requires the user to build a catalog of enumeration, which can serve later as meta-data ranges when learning queries. Meta-data ranges should consist of a set of homogeneous data elements such that a variable can be allowed to range over them. If not provided, defaults may be built by grouping homogeneously typed siblings in the schema into meta-data ranges.

- L_1: Foreign Key Relationships

This kind of data set requires the user to identify foreign key relationships between tables, or between tables and meta-data ranges. If not provided, reverse engineering of existing databases, or syntactic analysis of database schema definitions can easily yield such information. This kind of data set can also be represented graphically in entity-relationship form summing up the database designer’s intent.

- L_0: Row Examples

Example instances can be of two kinds: weak and strong. Both kinds of instances are associated with rows of the output table.

Definition: *Strong Example*

Higher order relational mappings, without group-by, relational difference and row ordering operators are monotonic. While an example instance for the concept of relational view definition is a set of input databases and an output view, it is possible to parcel out such an instance into smaller atomic instances. A *strong example* instance is a row in the output view of a relational mapping, with the corresponding variable assignment that produced it.

Definition: *Weak Example*

Weak Examples are associated with the concept of a materialized view and a *weak example* instance is simply a data row from the view. Weak examples are more ambiguous than strong ones since they only state that existing input tables can produce a given row, but does not specify which objects in the input are responsible.

One feature of strong examples is that they can potentially be completely hypothetical. They can answer questions of the form: if these rows belonged to the input databases, would this row be in the output. Adopting strong example for user interaction will significantly alleviate learning complexity by unambiguously specifying projection operators. Arguably strong examples are easier to understand and verify for the user than weak examples, since they contain their own intuitive explanation. Since strong examples are also conveniently richer in information, it makes sense in our active learning approach to use strong examples exclusively. Sphinx uses strong examples.

We note that in a different context, weak examples correspond to a more standard way to do machine learning since they allow automation. Strong examples almost require that a user be there to validate not only the output row, but also the input values that produced it. This kind of information is unlikely to be available in an automated approach. Conversely, weak examples drastically reduce the user input necessary, and a

system would be able to learn from a much larger training set. Also weak examples could be fed from an existing federated database if one already exists, thus allow the system to learn the corresponding definitions in an automated fashion. On the other hand, by adopting strong examples, we put a high burden on the user by precluding automation, but we hope to curtail the learning phase through active learning and by exploiting the richer and more explanative nature of strong examples.

1.6. User Interface

Richer data sets may form the basis for accurately learning richer concepts, but there is a cost in the complexity of user interface operations necessary to build those data sets. The worst interfaces will put the onus of defining relational queries entirely on the user, and will have no learning system. A typical SQL interface fits in that category and can hardly be called user friendly. The best interfaces will solicit the minimal amount of input necessary for an accurate definition and will do so in a manner familiar and intuitive for the user, leaving a learning system to do the rest.

- GUI_0: Very easy to specify

This category designates input that can be given through an operation that is both simple and intuitive such as a single click on a data element, a menu choice, or a drag and drop mouse action.

- GUI_1: Easy to specify

This category includes intuitive input that can be given through a sequence of simple operations. For example constructing an initial example by dragging and dropping data values as shown in Figure 3 will fall in this category.

- GUI_2: Repetitive

This includes intuitive input that can be given through a sequence of simple operations. However the number of operations approaches the size of a data dimension. For example, manually specifying an entire glossary or thesaurus to solve a domain/unit

conversion problem falls into this category, since at least one operation is necessary per entry.

- GUI_3: Abstract Operator

This includes input of a mouse driven nature, but that represents an abstract rather than an intuitive data manipulation. For example asking a user to click on an icon representing an operator requires knowledge of a related abstract concept, as opposed to clicking on a visualized data value. In most cases, such operators require the user to have knowledge of basic set theory and logic: for example knowing that some value is derived by averaging or some other value (aggregation). Another example is specifying arbitrary predicate by clicking on ">" or even typing in something like ">13". Thus we intend to regroup here partial specification of abstract concepts such as: aggregation, sorting, arithmetic, etc…

- GUI_4: Abstract Expression

We include in this category input that is not only abstract in nature, but also requires the user to learn specific syntax in order to form an accurate input expression. In particular, when a partial specification is not sufficient to narrow down the target concept to a finite or tractable number of possibilities, an actual expression or language fragment has to be specified. This includes input which has to be specified in a spreadsheet fashion or may even require the user to master some kind of expression language or even enter SQL fragments. In order to explicitly and completely specify an arbitrary arithmetic predicate such as "*x.price+x.tax>13*", a user must know an expression syntax well enough to specify both variables and attributes.

1.7. Interaction Chart

Parts of this chart are justified by the proof for the Graphic Input and Tractability theorems, which appear in Chapter 5. The rest of this chart represents reasonable expectations of input requirements for the various aspects of the learning problem.

1.7.1. Data Set vs. Complexity Chart

Vertically from top to bottom we move along the syntactic complexity ladder. At each step, when we go down, we assume that the feature described at any given row is excluded in all the rows beneath it. This chart is intended to summarize the conclusions, which can be drawn from our taxonomy. It is not a complete chart: most cells are left blank and it is impossible to exclude that in the future new or better learning algorithms could be invented to fill those blanks.

	L_4	L_3	L_2	L_1	L_0
Union					X
Embedded					
(none of the above)					
Unit and Domain Conversion[1]	Z			X	
Aggregation[2]		X, Y			X
Group-by[2]		X, Y			X
(none of the above)					
Self-Join[3]					X
Hidden Variable[4]				Y	X, Y
(none of the above)					
DNF					X
k-DNF					X
1-DNF with negation					X
(none of the above - 1-DNF only)					
Arbitrary Predicates					
Comparison Predicates					X
Equality Predicates[5]			X, Y, Z		X, Y, Z

X, Y and Z are meant to represent progressive degrees of achievement for each kind of input:

- Blank: this data set has no impact on learning or learning remains impossible
- X – data set necessary for a learning algorithm
- Y – data set necessary for a finite search space (learning algorithm will have less bias and more scope)

- Z – data set necessary for systematic convergence of a version spaces (bias-free exploration is reasonably possible)

  (1) This functionality can be automated given a well-documented schema and a proper example base [CB97], [PHC95], [DDH01], [DLD04].

  (2) Depending on the amount of information given by the user on the existence and nature of applicable group-by and aggregation operators, learning can proceed with varying degrees of completeness and scope. Grouping and ordering cannot be determined without row ordering examples. Learning an aggregation query means determining: the nature of the operator, the aggregation dimension and the applicable filters.

  (3) Self-joins are rare but sometimes necessary. While theoretically self-join opens up a huge search space, in practice, it is not realistic to learn queries with more than one self-join. Thus a biased approach to learning some self-joins could be elaborated.

  (4) In the absence of self-join, even in the worst-case scenario, hidden variables only provide join along finite join paths. Thus even the most exhaustive search space would be finite. Providing schema information would drastically reduce the size of the search space by limiting join paths within the scope of an entity-relationship diagram.

  (5) Meta-data, enumeration and typing data sets would allow compiling, executing and learning target queries in a second-order language such as SchemaSQL. Without such data sets, learning must be confined to first order SQL or Datalog.

1.7.2. GUI vs. Complexity Chart

This chart shows the projected impact on the learning system of the level of interaction required from the user. The more complex and the less intuitive the interface, the more complete and accurate the system can hope to be.

	GUI_4	GUI_3	GUI_2	GUI_1	GUI_0
Union[6]	X				X
Embedded[7]	X				X
(none of the above)					
Unit and Domain Conversion[8]	Z	Y	X, Y		
Aggregation[9]		Y, Z		X	X
Group-by[9]		Y, Z		X	X
(none of the above)					
Self-Joins[10]				X, Y	X
Hidden Variables[10]				X, Y	
(none of the above)					
DNF[11]				X	
k-DNF[12]				X	
1-DNF with negation[12]				X	
(none of the above - 1-DNF only)					
Arbitrary Predicates[13]	Y, Z	X		X	
Comparison Predicates[14]		Y, Z		X	
Equality Predicates				X, Y, Z	

X, Y and Z are meant to represent progressive degrees of achievement for each kind of interaction:

- Blank: this input has no impact on learning or learning remains impossible
- X – data set necessary for a learning algorithm
- Y – data set necessary for a finite search space (learning algorithm will have less bias and more scope)
- Z – data set necessary for systematic convergence of a version spaces (bias-free exploration is reasonably possible)

(6) Learning several queries separated by a relational union can be done one at a time, if the user to understand the concepts behind this ‘seeded’ learning.

(7) Learning several intermediate queries and re-using the materialized views as source data is possible, provided that the user understands the concepts behind this type of learning.

(8) Building a conversion thesaurus by hand is labor intensive. Complete specification of complex mappings, which may combine several fields, and

join paths, will require writing out a true relation expression expression ([DLD04]).

(9) Inside spreadsheet interfaces, aggregation and group-by operators are routinely specified by users.

(10) Handling self-join and hidden variables will add some complexity to any search space. Each hidden or self-join variable may be used as a filter. Thus, systematic learning is doubtful since in addition to expanding the number of possible joins, the number of possible selections is also trebled.

(11) Disjunctive concept acquisition ([Sebag96][NH93][KQ03][Smirnov01][Murray87])

(12) The size of the search space is not trivial even for simple languages. Nonetheless, given some biased learning is possible [Haussler88].

(13) Some arbitrary predicates must be specified fully, otherwise inductive learning will be limited to small, incomplete classes of target concepts.

(14) A simple hint such as a”>” will allow learning to proceed with a reasonably complete search space. However there are algorithms that learn real-valued attribute languages without such hints.

2. Related Systems

2.1. CLIO

The CLIO learning system is described in [YMH01]. Its position in the taxonomy is the following. It allows the learning for a single relational S/F/W clause. It does not allow meta-data ranges since it is a first order system. It allows Hidden joins and Self-joins. It assumes the Where line to be a conjunction of arbitrary predicates specified externally by the user and it does not address the issue of learning them.

The user can visualize the output of the system by browsing both the resulting tables, and by going through a set of “sufficient illustrations”. The learning is unsupervised and the user chooses and constructs new examples to submit to the system.

This process stops when the user is satisfied the result obtained is correct. The burden of verifying that Clio has learned the correct join path is entirely on the user, and that verification is a semi-recursive process: examination of all "sufficient illustrations" is not in itself sufficient to guarantee Clio has chosen the correct federating query. If, however an inconsistency is discovered by an astute user, a set of fine-tuning operators allow the user to correct it bringing Clio a step closer to the correct federating query. To that effect Clio requires the user to learn a new paradigm (a set of operators) to help it find the correct result.

2.2. LSD

LSD is a schema matching tool developed at the University of Washington [DDH01]. Whereas Sphinx assumes that the schema matching task is resolved locally by allowing the user to specify a data mapping instance, on the other hand LSD proposes to pit two databases with their respective schema or ontologies, and use both data and meta-data elements to propose schema correspondences. In addition to 1:1 mappings such as Sphinx assumes, LSD proposes to identify more complex mapping functions such as 1:m mappings as well as some composition or partition functions. Thus LSD and Sphinx essentially address orthogonal issues. Sphinx assumes that the schema matching has already been resolved and proposes to discover additional semantics, namely join and filter predicates as they apply to a federating query. LSD on the other hand assumes that two existing source ontologies are given, and proposes to discover the schema matching between them using machine learning techniques.

The similarity of schema elements and their eventual matching is made possible by a combination of two measures: structural and linguistic similarity. Linguistic similarity is based on the analysis on the domain, instances and label of schema elements. Structural similarity is based on the hierarchical positioning of a schema element in its source ontology.

The main challenges that LSD must overcome are Domain and Unit conversion operators, as well as partition or combination operators, when one schema element is

mapped to several elements (1:m mapping). Such is often the case with address fields which can be decomposed into smaller fields (street, state, zip …) or combined into a single large field.

Rahm and Bernstein present a survey of existing semi-automated schema matching tools. Such tools exploit both linguistic and structural features in order to score the similarity of matching elements between heterogeneous ontologies. The focus of those systems is similar to LSD and address issues of schema matching and unit and dictionary conversion issues (those systems are surveyed in [RB01]).

2.3. QBE Based Graphic Interface

A quick look at the SQL query wizard for a major commercial databases system allows us to find it a place in the taxonomy. The standard graphic interface is not a learning system and the user must specify every detail manually. However graphic interfaces place certain limiting assumptions on the generality of the queries that can be handled. A typical graphic interface handles single S/F/W SQL clauses SQL. Aggregation, group-by and sorting are handled post-facto by annotating query results with symbols. Typically unit conversions are not handled. Meta-data ranges are not allowed and there are no higher order features. Hidden joins and self-joins are allowed. The Where line is defined as a conjunction of arbitrary predicates specified by the user, and disjunction is not directly handled. A SQL syntax interface serves as a backup for complex queries.

There is a reasonable bias in most commercial applications. Clearly, for any query, there are a vast amount of possible predicates that can be built from variables and constants. However, only a tiny portion of those are ever likely to be present in report building queries for which those interfaces are designed. Thus, the framework of the query is built easily with graphical input taking care of joins and projections. More knowledgeable users can manually add additional filters if necessary.

3. Higher Order Languages

In [LSS97] and [LSS96] Lakshmanan et al. define SchemaLog and SchemaSQL as query languages respectively derived from Datalog and SQL. These two languages allow the higher order syntactic constructions necessary for schema transformations as suggested by Krishnamurthy, Litwin and Kent [KLK91]. Andrews et al. outline an implementation for SchemaLog [ALS96]. This implementation relies on modifying a first order query engine and incorporating the operators necessary to deal with the higher order features of the language. A scheme to implement SchemaSQL by compilation is first introduced ([LSS96]), then later a modified SQL engine directly capable of executing SchemaSQL [LSS99]. Miller introduces a range of applications suggesting there may be an appetite for using a language like SchemaSQL to provide certain database services in addition to defining federated views [Miller98].

HiLog ([CKW93]) is a higher order logic programming language but lacks some of the syntactic properties of SchemaLog and is not envisioned for databases applications. Issues regarding its main memory implementation are studied in [SW95].

In a mediated architecture the specification of data manipulations that federate data sources are exploited by mediators to resolve the problem of distributing and optimizing multi-database query answering. The Local-As-View approach (LAV) specifies the tables of existing sources as a query on the federated schema. The Global-As-View approach (GAV) specifies the federated schema as a set of views defined over the source databases. LAV and GAV are inverse from each other, and though theoretically equivalent approaches, in practice they have different advantages [FLM98]. Vassalos and Papakonstantinou introduce a higher order language termed RQDL suitable for describing data integration inside a mediated architecture [VP97]. They describe a translation process of that language into a first order logic language noted p-Datalog, with function symbols. In [ACM96], Abiteboul, Cluet and Milo, introduce a system suitable for integrating object-oriented databases. They introduce a logic language capable of describing some tree transformations for object-oriented schema integration. In addition

to these efforts, a vast amount of mediated systems were proposed ([GMP97], [HKW97], [VRG98] among others) and catalogued by Florescu et al. [FLM98]. The UniSQL language is described in [KGK95] and introduces operators to process dynamic schema updates. The SIMS system proposed by Arens et al. (see [AK93], [AKS96]) introduces a query rewriting architecture for federating heterogeneous databases.

4. Version Spaces Candidate Elimination Algorithm

Given a set of training examples, both positive and negative, and given a concept class C, Mitchell's version spaces inductive learning algorithm proposes to maintain and update a hypothesis space comprising all elements of C consistent with the training data. Version spaces requires a partial order in C. By default a partial order relation in C, *generality*, can be defined. For two concepts c and c' in C, c is *more general than* c' when the set of positive examples covered by c' is included in the set of positive example covered by c.

To keep the hypothesis space consistent with positive examples, we can define a maximally general boundary set $G \subseteq C$. Concepts that are strictly more specific than any element of G are excluded from the hypothesis space. Similarly, a maximally specific boundary set S, derived from negative examples, helps exclude concepts that are strictly more general than any element of S. The concept class C is *admissible* if either S or G can always be reduced to a single element, i.e. the partial order admits either a least upper bound or a greater lower bound.

The version spaces algorithm is said to have collapsed when the hypothesis space is empty (i.e. S is more general than G). It is said to have converged when the hypothesis space is reduced to one element (S equals G). It is said to be exhausted when it is either collapsed or converged.

The original version spaces or candidate elimination algorithm provides three key features for inductive learning: two distinguished hypothesis sets S and G each with a well-understood bias with respect to the partial order defined for C, and a description of

all remaining consistent hypothesis. In our work we mainly use the third feature, to drive active learning and seek convergence: our concept class (1-DNF) is sufficiently small to allow this.

Unless C is chosen very carefully, version spaces may not represent a realistic approach ([Haussler88]). Even given an admissible concept class C, the remaining one of the two boundary set descriptions may not be tractable. To overcome this serious limitation, Hirsh et al. proposed instance-based boundary sets [HMP97], [Smirnov01]. Rather than being a set of boundary hypotheses, one of both of the set S and G becomes a set of example instances. Key properties of the version space algorithm are preserved, and tractability for larger class of concepts is improved. Smirnov et al. proposes a hybrid form on boundary sets, adaptable boundary sets [SHS02].

While the practicality of version space was originally thought to be limited to noise-free data and conjunctive concepts, new algorithms using version spaces have been developed to deal with noisy data ([NH93]) and to learn disjunctive concepts [Michalski83], [Murray 87], [Sebag96], [KQ03]. The latter disjunctive algorithms, may use Mitchell's version space operators, but do not adopt the original version spaces approach: they do not keep track of a complete hypothesis space and do not address any issue of convergence. All that is output by those algorithms is a set of distinguished hypothesis forged with a particular implicit or explicit bias.

Chapter 3 Sphinx Prototype Architecture

In line with our overall goal to work towards providing database federation of a more casual nature, when it comes to building a federated database system our goals must be twofold:

- Minimize the burden of cooperation on each component database
- Minimize the cost of adding new heterogeneous platforms

There are two recognized approaches to build a federated database system: the mediated query answering approach, and the data warehousing approach [Widom96]. Our architecture materializes a non-mediated system, exploiting completely independent databases with no modifications, and ODBC-type connectivity. A traditional mediated approach requires the architecture to provide a distributed query engine. Mediators optimize and distribute queries down to the component databases, and integrate the subsequent results with merging and post-processing. These global optimization considerations require interaction between component databases at the catalog level and cross-database relational joins require distributed algorithms. Individual data sources are independent entities and requiring high-level cooperation pose serious organizational challenges. On the other hand an ODBC type of connectivity requires as the sum of all necessary cooperation, the opening of a SQL-enabled data transfer port on the database. Significantly, such connectivity is at the data level, and it is the case that all meta-data and catalog access commands are broadly incompatible from one platform to another. Thus, adopting a data warehousing rather than mediated approach enables us to maintain a low degree of cooperation combined with a high degree of portability.

Contemplating a data warehousing approach is done with no restriction to the generality of our solution, but 1. it diminishes the number of implementation details we

need to consider 2. we abandon any claims on cross-platform optimization 3. we assume the migration of data to a common platform. Thus this architecture is not suitable for the seamless integration of databases through a mediated and distributed query answering system. However the architecture will deliver an effective and extremely portable solution for the migration of data from source to federated schema, by means of federating queries implemented in a data warehousing context.

Another characteristic of our approach is that we do not materialize any query engine internals. Rather we rely on the existing native query engines of each source database, their operators and their optimizers, as the sole providers of relational data processing. Since almost all existing commercial database platforms offer SQL connectivity and now ODBC and JDBC as well, our approach ensures portability regardless of the heterogeneous nature of the database components.

1. Architecture Components

Our solution must address the two main challenges in allowing a user to graphically specify data federating queries: overcoming ambiguity and efficiently handling a more expressive second order specification.

The four components of our solution are shown in Figure 6:

- Preprocessing: A data and meta-data preprocessing module.
- Graphical Interface: An easy to use graphic user interface (GUI) for the learning system.
- Learning Algorithm: A second order query learning system
- Execution Engine: An execution module for higher order queries built on top of the standard SQL engine.

In order to build a complete Federated Database by Example system, this architecture combines solutions to three different challenges:

- Reverse engineer the existing databases by exploiting their catalogs to populate a central catalog.
- Enable the user to lend intelligence to the system by expressing the desired result, and transcribing it into an unambiguous semantic specification (in SchemaSQL).
- Execute the semantic specifications on the data with an efficient execution mechanism for second order SchemaSQL.

Drawing on practical experience from integrating databases, we notice that the elements of our architecture happen to correspond to the major steps involved in solving integration problems.

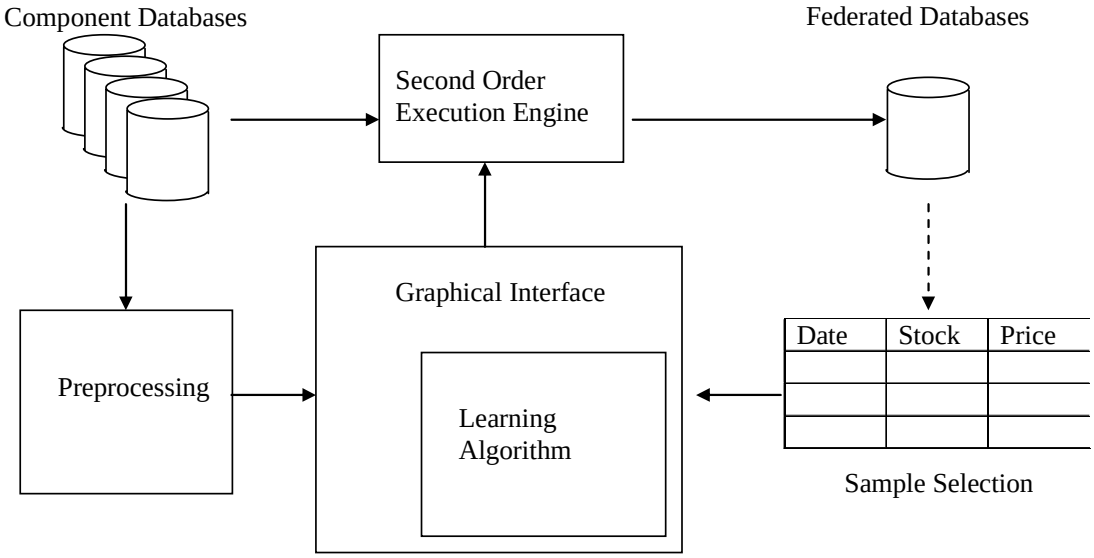


Figure 6 – Architecture

The first step is information gathering, both about issues related to the data domain and to the catalog content of the local databases. This is a crucial phase accomplished by engineers drawing on domain knowledge. The gathered information aims to take a census of the existing meta-data catalogs and to label appropriately each element with characteristics and constraints. This essentially amounts to reverse engineering the local databases in order to understand how they can be federated.

At the other extremity of the process chain are the transformation solutions. Wrapper, mediators and query engines are charged with transforming the data in preset or programmable ways. These physical transformations are the 'robotic' assembly lines, which deliver the finished products of the data warehouse or integrated system. Underlying the whole integration process is a logic that provides for the semantic integration of the data. It draws on the knowledge gathered at the beginning of the process and proceeds to direct the transformation mechanism to achieve a designed result.

2. Preprocessing

The first job of the Preprocessing layer is to identify the different component databases and extract their catalogs. With this data, the graphic interface will be able to offer a logical representation of the sources to the user.

Once the scope of all the entities in local sources is clear, the system can reverse-engineer some of the underlying relationships between those entities. The preprocessing system will explore the databases for functional dependencies and gather catalog statistics. These dependencies and statistics describe the data. We will see later how dependencies play a role in creating missing examples. Knowledge of some dependencies and catalog statistics are exploited to allow a heuristic exploration of the search space during the active learning phase.

3. Graphical Interface

The role of the user input section is to capture the user's practical knowledge of the data without forcing him to use a formal syntax to state that knowledge. We aim to require the least possible effort from the user in terms of completely specifying the Federated schema. Active learning will also provide positive and negative output examples to the learning system and proceed through the interface by the straightforward submission of examples taken from the data.

4. Learning System

The goal of the learning system is derive an unambiguous specification of the query initiated by the user through the graphic interface. The learning system will explore a Hypothesis space of all basic project-select-join transformations consistent with the user given examples. Using the Query Execution module, the learner will produce positive and negative examples to submit to the user. Each additional interaction with the user will let the system further discriminate among the remaining plausible hypotheses. Eventually the learner will know when it has converged to a result and output a final query, without requiring further validation or review by the user. The verification has taken place solely through the process of example-based interaction, with examples chosen by the system.

5. Execution Engine

The system must be capable of executing over the component databases any second order query in the learning system. The execution engine will be used to produce examples taken from the data to help the active learning as well as delivering the final federated database tables once they have been defined.

Thus, it must be made possible to execute complex and potentially second order queries efficiently over the data sources. Our goal is not to implement a native second order query engine, but to define a compilation of second order queries into first order SQL or Datalog queries. The resulting compiled program must then be executed and efficiently optimized by existing database engines.

Chapter 4 SchemaSQL Execution Engine

Building on Krishnamurthy et al. ([KLK91]) concept of a higher order query language as a possible syntactic extension of Datalog, coined IDL, Lakshmanan et al. introduced SchemaSQL ([LSS96]) and SchemaLog ([LSS97]) as higher order query languages suitable for relational databases. A compilation scheme was proposed for SchemaSQL, and a modified execution engine was built for both SchemaSQL and SchemaLog ([LSS99]).

For the purpose of building a federated database environment with Sphinx, it becomes necessary to provide an implemented solution for SchemaSQL as part of the architecture. The first part of this work focused on learning a SchemaSQL query, the next part must focus on the issues of dealing with a second order language when executing a SchemaSQL query. We saw that the SchemaSQL execution engine element of the architecture provides two roles: it allows the querying of source data for sample selection, and it provides the means to move the data once the federated queries have been specified.

Two kinds of implementations have been proposed for higher order languages: compilation of into an existing language such as SQL or execution by a specialized engine. Building such an execution engine can be done by modifying and enhancing an existing database query engine. The importance of implementing an efficient compilation for SchemaSQL derives from two key observations consistent with our architectural goals. The first is that compilation does not require modifying an ordinary relational query engine. This is appropriate when the goal is to federate data but otherwise exploit the existing RDBMS engines and services through an API such as ODBC. Such interfaces are minimally burdensome ways for data owners to cooperate by offering

access to their repositories. Further, compiling is a portable approach well suited to implementing queries on heterogeneous systems. Thus our chosen architecture suggests using a simple compilation process for higher order queries. In particular, a compiler is easily inserted between the specification of higher order queries by Sphinx and the relational engines that are charged with executing them.

By compiling SchemaSQL to SQL, our method enables the development and execution of higher-order query languages in a layered architecture where the impact of higher-order operators is isolated and rendered compatible with existing query engines.

We define a compilation of SchemaSQL into standard SQL. We show that the compiled output is of size O(m+p) where m is the size of the catalogs and p the size of input queries. The resulting code may be executed by existing conventional SQL query engines without modification. We extend our basic compilation method by including type driven optimizations, which empirical evaluation shows, yield an effective execution by native query engines. Prior efforts do not provide feasible guarantees on the size of the compiled programs or require the development of new query engines encompassing higher-order query operators.

The construction of our target code amounts to carefully substituting the meta-variables with strings from the catalog. The bound on target size is achieved, in part, by noting the primary keys in the source databases and exploiting the uniqueness of those keys to sidestep multiplicative aspects of blind substitutions.

The implementation we report on includes static optimizations using data types and schema elements. Further, after additional basic semantic optimizations, it encapsulates higher order operators into simple SQL joins such that a query engine executing the target code can optimize them. The space bounds on the size of the target code are a useful if not necessary guarantee on the nature of the compilation. Even so, the target code may contain multi-way joins, which if executed using poor query-plans may prove expensive. We show that by including static optimizations, our compiler produces target code, which is executed effectively by a standard SQL engine.

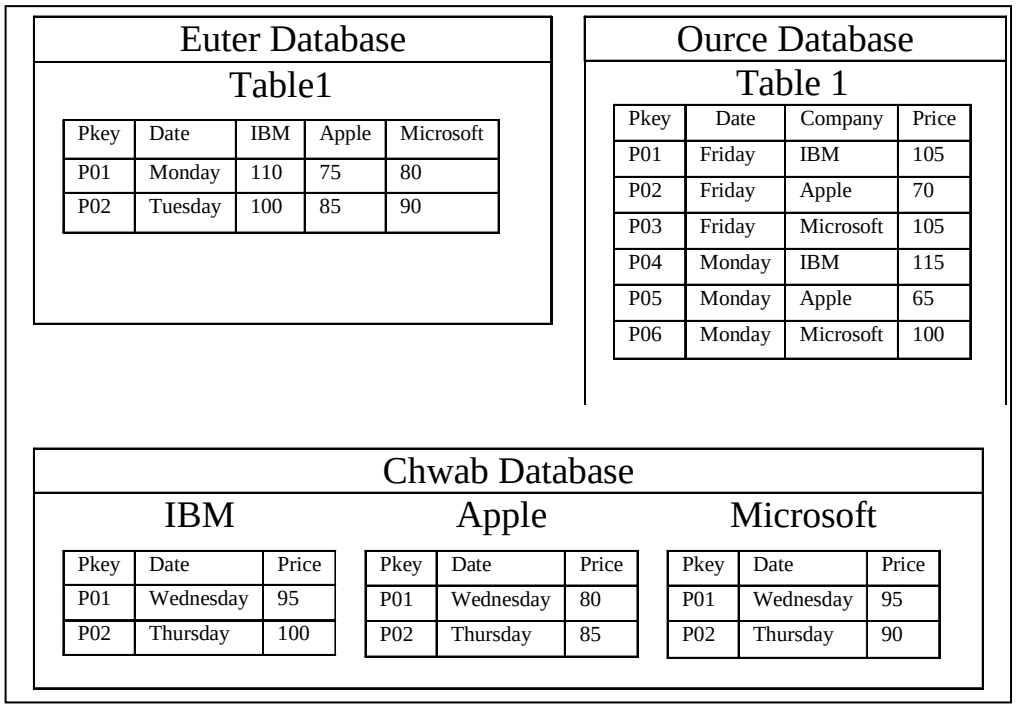


Figure 7- Krishnamurthy's stock market databases

```
Q5:   Select (T1.Company, T2.Company, T3.Company)
      From   Ource::Table1   T1
             Ource::Table1   T2
             Ource::Table1   T3
      Where  T1.Price>T2.Price+T3.Price
             And T2.Price > T3.Price
             And T1.Date=T2.Date=T3.Date
             And T1.Company <> T2.Company
             And T1.Company <> T3.Company
             And T2.Company <> T3.Company
```

```
Q6:   Select (T1.Company, T2.Company)
      From   Ource::Table1   T1
             Ource::Table1   T2
      Where  T1.Price > T2.Price
             T1.Date = T2.Date
```

Figure 8 – Standard SQL queries over the Ource Database

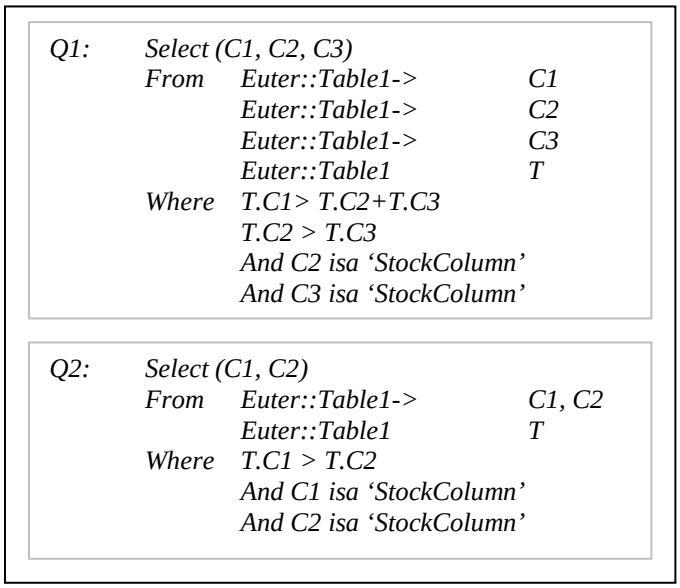


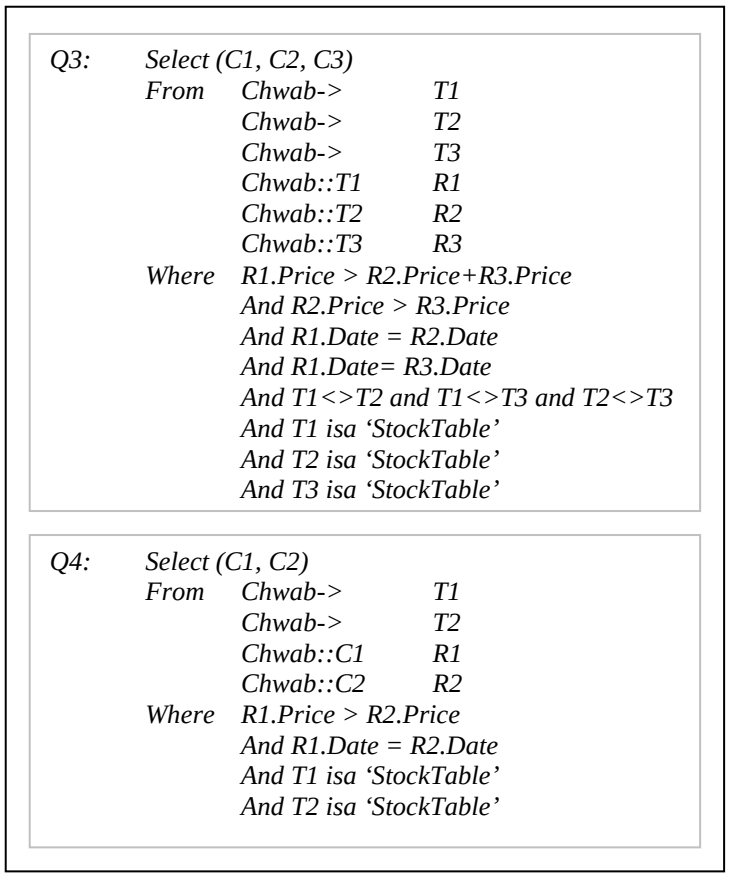
Figure 9 – SchemaSQL queries over the Euter Database

We adopt the running example shown in Figure 7, introduced by Krishnamurty et al. to illustrate the challenging nature of higher order languages. Figure 8 illustrates two queries over the database Ource. While these two do not represent meaningful queries, they were chosen because they contain multiple meta-data variables in the same Select/From/Where statement, allowing us to illustrate our compilation and the resulting complexity of the compiled program.

The first query returns all stock triplets such that for some date, the price of the first stock is greater than the price of the other two combined. The second query returns all pairs of stock, ordered by their price on a given date. These queries are written in the usual SQL syntax for Database Ource in Figure 8. The syntax "*From Ource::Table1 T* " denotes that T stands for a row in Table1, in database Ource.

The same queries over databases Chwab and Euter cannot be written out in SQL. In order to express these queries concisely, SchemaSQL must be used. Variables are allowed to be defined and to range over database tables or table columns. In Figure 9, Q1 and Q2 are expressed using C1, C2 and C3. In SchemaSQL, the expression "*From*

Euter::Table1 C“, marks C to be a variable ranging over the set of columns of table Euter::Table1. The expression “*C isa ‘StockColumn’”* is a selection predicate, which ensures that all values that C can range over are of uniform type. Practically type ‘StockColumn’ is the set: {“Euter”, “Ource”, “Chwab”}. In Figure 10, the same two queries are expressed over the database Chwab using T1, T2 and T3, variables ranging over the set of tables: {“Euter”, “Ource”, “Chwab”} which form the type ‘StockTable’. Note that the syntax “*From Chwab-> T”* , indicates that T is a variable ranging over the tables of database Chwab.



```
Q3:   Select (C1, C2, C3)
      From   Chwab->        T1
             Chwab->        T2
             Chwab->        T3
             Chwab::T1      R1
             Chwab::T2      R2
             Chwab::T3      R3
      Where  R1.Price > R2.Price+R3.Price
             And R2.Price > R3.Price
             And R1.Date = R2.Date
             And R1.Date= R3.Date
             And T1<>T2 and T1<>T3 and T2<>T3
             And T1 isa ‘StockTable’
             And T2 isa ‘StockTable’
             And T3 isa ‘StockTable’

Q4:   Select (C1, C2)
      From   Chwab->        T1
             Chwab->        T2
             Chwab::C1      R1
             Chwab::C2      R2
      Where  R1.Price > R2.Price
             And R1.Date = R2.Date
             And T1 isa ‘StockTable’
             And T2 isa ‘StockTable’
```

Figure 10 – SchemaSQL queries over the Chwab database

1. Query Compilation

Compiling is done in three steps: creation of intermediate views over the source database, and translation of higher order queries into standard SQL and optimization of the results. The intermediate views are an "unskolemized" form of the source database's system catalog and other meta-data such that a data element is generated for each meta-data element . The compiler then substitutes meta-data variables in the input program with first-order variables ranging over the Skolem constants. The term "unskolemization" was introduced for a similar construction in automatic theorem proving [BM75].

As a starting point, every table, in every database, is assumed to contain one column, which can be used as a primary key. In our running example, we introduced one column, named Pkey, to each table to serve this purpose (see Figure 7). Our compilation leverages primary keys in order to rewrite expressions containing meta-data variables into joins. This allows standard SQL engines to optimize complex expressions mixing several meta-data variables, as a normal set of join expressions.

For the purpose of this exposition, without loss of generality we will assume that there are no database variables involved in the SchemaSQL queries submitted to the compiler.

1.1. Defining the Intermediate Views

There are two kinds of intermediate views. Both kinds exploit the notion of types in their construction. In SchemaSQL, meta-data variables can be typed by predicates such as "*var isa TypeName*". Those meta-data types can be viewed as enumerations of homogeneous catalog elements that meta-data variables are allowed to range over. If those types are not defined by the user, they must be inferred by default. In the default typing system, the type of a column is inferred from its SQL type: the columns of each table that are of identical SQL type form an enumeration. Column variables can range over those enumerations. The type of a table is defined by the juxtaposition of the type of its columns. All the tables of identical type in a given database can form an enumeration

that table variables are allowed to range over. For more details on types in SchemaSQL, see [LSS96].

The first kind of intermediate view is defined by augmenting each table with its corresponding catalog information. Database and Table elements are mapped into data fields using Skolem constants to represent them with the following algorithm.

Algorithm 1.1a (Creation of TypeTable Views).

- For every table type TypeTC such that a table DB::T in *TypeTC* has n+1 columns (Pkey, Col_1, Col_2, …, Col_n):

  Create View TypeTC ('Database',' Table', 'Value$_1$',' Value$_2$', …, 'Value$_n$',' Key')

  As Union [U_{TC}]

- For every table DB::T in TypeTC (Pkey, Col_1, Col_2, …, Col_n) insert into Union U_{TC}:

  Select ('DB', 'T', r.Col$_1$, r.Col$_2$, …, r.Col$_n$, r.Pkey)

  From DB::T r

This intermediate view is defined by creating one row per relevant table in the source databases. Its purpose is to help retrieve row values defined in expressions such as "*From DB::T row*" when T is itself a variable. This algorithm applied to type *StockTable* is shown in Figure 11.

Note that in this construction the Skolem constant introduced for table T, is simply the string T containing the name of the table. In certain cases, when all the tables within a database are of the same type, such as in database Chwab, the syntax allows the selection predicate isa(type..) to be omitted. As we discussed earlier, in such an instance, the

compiler has to create a default type that will contain all the tables in it.

View TypeStockTable:

```
Create View TypeStockTable (database, table, value1, value2, pkey) AS
Select ('Chwab', 'IBM', r.Date, r.Price, r.Pkey )
From    Chwab.IBM       r
Union
Select ('Chwab', 'Apple', r.Date, r.Price, r.Pkey )
From    Chwab.Apple     r
Union
Select ('Chwab', 'Microsoft', r.Date, r.Price, r.Pkey )
From    Chwab.Microsoft r
```

Figure 11 – Intermediate View for TypeStockTable (comprising Chwab::IBM, Chwab::Apple, Chwab::Microsoft)

The second part is the creation of intermediate views for column types. For every column type CC a similar algorithm has to be applied.

Algorithm 1.1b (Creation of TypeColumn Views).

- For every column class TypeCC, such that there is at least one column C of type TypeCC:

  Create View TypeCC ('Database', 'Table', 'Column', 'Value', 'Pkey')

  As Union $[U_{CC}]$

- For every column C of type TypeCC, such that C appears in table DB::T insert into Union Ucc:

  Select ('DB', 'T', 'C', r.C, r.Pkey)

  From DB::T r

This last intermediate view is defined by a creating one row for every relevant cell in the source tables. Its purpose is to help retrieve the proper value of expressions involving column variables such as: row.colvar. The value of such expressions can be retrieved from the appropriate row in the TypeCC view by selecting on the Database,

Table, Column and Pkey fields. Figures 11 and 12, respectively, illustrate the algorithm for a *StockTable* and a *StockColumn* meta-data type.

View TypeStockColumn:

```
Create View TypeStockColumn (Database, Table, Column, Value, Pkey) As
Select ("Euter", "Table1", "IBM", r.IBM, r.Pkey)
From    Euter.Table1    r
Union
Select ("Euter", "Table1", "Apple", r.Apple, r.Pkey)
From    Euter.Table1    r
Union
Select ("Euter", "Table1", "Microsoft", r.Microsoft, r.Pkey)
From    Euter.Table1    r
```

Figure 12 – Intermediate View for TypeStockColumn (comprising IBM: int, Apple: int, Microsoft: int)

1.2. Generating the First-Order Queries

The second step is to rewrite the higher order queries in the program to be compiled. Those queries must be rewritten so as to derive their variables from the appropriate intermediate view whenever an expression involving a meta-data variable appears. Meta-data variables will no longer appear in the rewritten queries since the necessary catalog information from the database sources has been incorporated into the intermediate views and can be accessed with regular variables. Thus the resulting queries become simple select/from/where statements in standard SQL.

The following algorithm takes each select/from/where statement appearing in the higher order SchemaSQL program and translates it into a standard SQL statement. In order to achieve this result, the principal substitutions that occur are:

- row variables ranging over a table that is defined by a table variable are substituted by a row variable ranging over the appropriate intermediate view for the table's type

- row.cv expressions, where cv is an column variable, are retrieved from the Value column of the intermediate view for that column's type by selecting on the Database, Table, Column and Pkey fields.
- table variables are substituted by a projection on the Table column of the appropriate view for that variable's type.

In every step of the following algorithm, "r_1" will represent a newly generated variable name.

Algorithm 1.2 (Higher-Order Compile).

Step 1.

- Execute for every row variable r

Input:

- In the From clause, r's range is *DB::T*, where T is a table variable of type TypeTC (col_1, col_2, … , col_n)

Action:

- In the From clause, replace *DB::T* with *TypeTC* as a range for r
- In the Where clause, insert as a selection "*r.Database = DB"*
- In the Where clause, insert as a selection "*r.Table = T"*
- In the Select and Where clauses, replace every *r.column* expression, with $r.value_i$ where column is the i^{th} column in DB::T

Step 2.

- For every expression r.cv where r is a row variable and cv is a column variable of type TypeC

Input:

- cv is of type TypeCC

Action:

- In the From clause, introduce r1 with Range TypeCC: "*TypeCC r1*"
- In the Where clause, insert as a join "*r1.Database = r.Database*"
- In the Where clause, insert as a join "*r1.Table = r.Table*"

- In the Where clause, insert as a join "*r1.Pkey = r.Pkey*"
- In the Where clause, insert as a join "*r1.Column = cv*"
- In the Select and Where clauses, replace every occurrence of "*r.cv*" with "*r1.col*"

Step 3.

- Execute for every column variable cv such that cv is of type TypeCC

Input:

- In the From clause cv appears with range *DB::T*, where cv is a column of type TypeCC

Case:

- If step 2 was executed at least twice for expressions of type "$row_1.cv$" and "$row_2.cv$"

Action:

- In the From clause, remove the entry for cv: "*DB::T cv*"
- In the Where clause, replace all expressions "$row_1.Column = cv$", "$row_2.Column=cv$", …, "$row_n.Column=cv$" with the n-1 following join predicates "$row_1.Column = row_2.Column = \ldots = row_n.Column$"
- In the Select and Where clause replace remaining occurrences of "*cv*" with "$row_1.Column$"

Case:

- If step 2 was executed exactly once for the expression "$row_1.cv$"

Action:

- In the From clause, remove the entry for cv: "*DB::T cv*"
- In the Select and Where clause replace remaining occurrences of "*cv*" with "$row_1.Column$"

Case:

- If step 2 was never executed for any expression of type "*row.cv*"

Action:

- In the From clause replace "*DB::T cv*" with "*TypeCC r1*"
- In the Where clause, insert as a join "*r1.Database = DB*"

- In the Where clause, insert as a join "*r1.Table = T*"
- In the Select and Where clauses, replace every occurrence of "*cv*" with "*r1.Column*"

Step 4.

- Execute for every table variable T such that T is of type TypeTC (col_1, col_2, … , col_n)

Input:

- In the From clause, T appears with range *DB*.

Case:

- If steps 1 and steps 3 combined to form two or more expressions of type "row_1*.Table = T*", "row_2*.Table = T*", … "row_n*.Table=T*" in the Where clause

Action:

- Replace with n-1 join predicates: "row_1*.Table=*row_2*.Table=…*row_n*.Table*"
- In the From clause remove the entry "*DB T*"
- In the Select and From clause replace every remaining occurrence of "*T*" with "row_i*.Table*", preferably where row_i is a variable over TypeTC rather than over a TypeColumn.

Case:

- If steps 1 and steps 3 combined to form exactly one expression of type "row_1.Table = T" in the Where clause

Action:

- In the From clause remove the entry "*DB T*"
- In the Select and From clause replace every remaining occurrence of "*T*" with "row_1*.Table*".

Case:

- If steps 1 and steps 3 never produced an expression of type "*row.Table = T*" in the Where clause.

Action:

- In the From clause replace the entry "*DB T*" with "*TypeTC r1*"

- In the Where, insert as a selection “*r1.Database =DB*”
- In the Select and Where clause, replace all occurrence of “*T*” with “*r1.Table*”

Figure 13 shows the application of Algorithm 1.2 to queries Q1 through Q6 in our running example. This compilation method is proven correct in a separate exposition ([BM01]).

```
Q1:   Select (R1.Column, R2.Column, R3.Column)
      From   TypeStockColumn        R1, R2, R3
             Euter::Table1          R4, R5, R6
      Where  R1.Database = R2.Database = R3.Database = 'Euter'
             And R1.Table = R2.Table = R3.Table = 'Table1'
             And R1.Value > R2.Value+R3.Value
             And R2.Value > R3.Value
             And R1.Column <>R2.Column
             And R1.Column<>R3.Column
             And R2.Column<>R3.Column
             And R1.Pkey = R4.Pkey and R2.Pkey = R5.Pkey and R3.Pkey = R6.Pkey
             And R3.Date = R4.Date = R5.Date
```

```
Q2:   Select (R1.Column, R2.Column)
      From   TypeStockColumn        R1, R2
             Euter::Table1          R3, R4
      Where  R1.Database = R2.Database = 'Euter'
             And R1.Table = R2.Table = 'Table1'
             And R1.Value > R2.Value
             And R1.Column <>R2.Column
             And R1.Pkey = R3.Pkey and R2.Pkey = R4.Pkey
             And R3.Date = R4.Date
```

```
Q3:   Select (R1.Table, R2.Table, R3.Table)
      From   TypeStockTable   R1, R2, R3
      Where  R1.Database = R2.Database = R3.Database = "Chwab"
             And R1.Value1 = R2.Value1 = R3.Value1
             And R1.Value2 > R2.Value2 + R3.Value2
             And R2.Value2 > R3.Value2
             And R1.Table<>R2.Table and R1.Table<>R3.Table and R2.Table<>R3.Table
```

```
Q4:   Select (R1.Table, R2.Table)
      From   TypeStockTable   R1, R2
      Where  R1.Database = R2.Database = "Chwab"
             And R1.Value1 = R2.Value1
             And R1.Value2 < R2.Value2
```

```
Q5:   Select (R1.Company, R2.Company, R3.Company)
      From   Ource::Table1    R1, R2, R3
      Where   R1.Date = R2.Date = R3.Date
             And R1.Price = R2.Price + R3.Price
             And R1.Company<>R2.Company and R1.Company<>R3.Company
             And R2.Company<>R3.Company
```

```
Q6:   Select (R1.Value2, R2.Value2)
      From   Ource::Table1    R1, R2
      Where  R1.Date = R2.Date
             And R1.Price < R2.Price
```

Figure 13 – Compiled Queries

2. Optimizing Intermediate Views

The intermediate view definitions created so far are lossless, resulting in much larger tables than necessary. Following is an optimization method useful when the views are to be materialized or the target query engine is incapable of optimizing queries over views. The optimization amounts to pushing predicates down from the generated SQL into the intermediate view definitions.

Algorithm 2 proceeds by examining every statement produced up to this point.

Algorithm 2 (Optimize Intermediate Views).

Step 1.

- Execute for all *TypeTable* views

Case:

- "*TypeTable row*" is a line in the From clause of compiled statement *sfw*,

Action:

- Create a new view using every selection predicate *SP(row)* appearing in statement *sfw*:

Create View TypeTableOptimized-sfw-row

As *Select **

From TypeTable t

Where SP(t)

- Replace "*TypeTable row*" with "*TypeTableOptimized-sfw-row row*" in the From clause of statement *sfw*

Step 2.

- Execute for all *TypeColumn* views

Case:

- If "*TypeColumn col*" is a line in the From clause of compiled statement *sfw*

Action:

- Create a new view using every selection predicate *SP(col)* appearing in statement *sfw*

Create View TypeColumnOptimized-sfw-col

*As Select **

From TypeColumn c

Where SP(c)

- Replace "*TypeColumn col*" with "*TypeColumnOptimized-sfw-col col*" in the From clause of statement *sfw*

Further, the SQL compiler should be directed to materialize these optimized views into tables whenever it is appropriate and the space is available. When possible, materializing will save on the number of run-time union operations performed by the query engine, which is always a good policy. The views should only be materialized right before a query; and there is an incurred cost, which can be negligible even for large queries as long as table and intermediate view sizes are reasonable.

3. Size and Complexity

To validate the feasibility and practicality of our compilation, two aspects of the generated output have to be considered. The first one is the size of the compiled program, measured for example as a character string. Size is a legitimate measure to determine the feasibility of executing the compiled output since it is directly related to the number of Select/From/Where statements generated and their size. Another aspect of the performance of the compiler is to measure the complexity of the output queries it generates. A compiler producing a small number of computationally intensive S/F/W

statements is just as inefficient as a compiler producing vast quantities of computationally simple S/F/W statements.

3.1. Size

Because higher order programs compiled into SQL must include meta-data catalog elements in them, the size of the compiled program will be some function of m, size of the meta-data catalogs. To be feasible a compilation must produce programs of manageable size and contain a limited number of select/from/where (S/F/W) statements. Obviously compilations yielding programs of size m^k, where k is allowed to grow arbitrarily with the input are unacceptable.

To measure the size of the output program we consider S/F/W statement to be made up of Select lines, From lines, and Where lines. The Select clause is considered to make up only one line. Every Range-Abbreviation combination in the From clause is also considered to make up one line. Finally, every predicate in the Where clause forms a line. Except for the select line, each line contains two expressions, and possibly a logical connective. This is the case for a predicate (i.e. A.b = c.d) or for a Range-Abbreviation combination (i.e. Customer c). Expressions are the unit in which the size of S/F/W statements will be measured here.

Theorem 3.1 (Size Guarantee). If the size of the meta-data catalog is m and the size of the input program is p, then the size of the compiled program p' is bound by:

$$p' < k*m + 16*p$$

and k is not dependent on the input, but is a constant factor of the compilation construction. k is the maximum size of a select/from/where statement inserted by Algorithm 1.1a and Algorithm 1.1b. Thus p' is linear in m and p, and asymptotically is a O(m)+O(p) and also a O(m+p).

Proof:

Algorithm 1.1a and Algorithm 1.1b create a set of intermediate views. The optimized version of those views produced by Algorithm 2 will not be considered here.

Creating a TypeColumn view requires one SQL statement of fixed size for all the columns in the TypeColumn data type. Assuming that every column in the heterogeneous databases belongs to exactly one data type, there is at most one SQL statement per column.

Size (ArityXTable creation for all X) < k2(# of Columns in the catalogs)*

Creating the TypeTable views requires one SQL statement for each table in the type TypeTable. The size of each of these statements is bound by a linear function of the number of columns in that table. Again, we will assume that each table in the heterogeneous databases belongs to exactly one type.

Size (TypeTable SQL statement for each table T) < k1(# of Column in T)*

$\Rightarrow$

Size (TypeTable view creations for all tables T) < k1(# of Columns in the catalogs)*

Thus the size of the statements generated by the first compilation step can be bound by a k*m, where $m = m_1+m_2+..+m_n$, and m_i is the size of the catalog for database i in the federation.

Algorithm 1.2 generates a standard SQL query for each input higher order query. Let p be the size of the input program (unit: number of expressions). Here are the worst-case bounds:

- For every row variable 1 line is removed and 3 lines are added (in Step 1.)
- For every distinct "*row.col*" row and column combination, 5 lines are added (in Step 2.)
- For every column variable, in the worst case 1 line is removed and 5 are added. (in Step 3.)

- For every table variable, in the worst case 1 line is removed and 3 are added (in Step 3.)

Note that Algorithm 1.2 never inserts a higher order variable or a new “row.col” row/column combination. Therefore those numbers are not cumulative. Also, each line inserted by Algorithm 1.2 contains at most two expressions and an equality sign. The size of the Select line is unchanged.

The worse case size increase is for row/column combinations, where every expression provokes the insertion of at most 5*(2 expressions + 1 equality sign). Thus a very gross linear bound for the output produce by the second step is p+15*p. (A more precise bound would be in fact much better).

The reader can verify that adding the optimizations described in Section 2 does not alter the size guarantee.

3.2. Run-Time Complexity

Our complexity measure is relevant insofar as we measure the complexity of the compiler output versus that of the compiler input. Thus we are only interested in the additional complexity due to the compiler’s work. In order to measure that quantity we will not examine precise cost functions relating to the complexity of each statement, but leave that evaluation work to the experimental Section. Instead, we identify here, each explicit join in the output program, subtract each explicit join, that was already present in the input program, and we obtain the joins “added” by the compiler.

The first step of compilation yields SQL statements that do not contain joins. In the second step, Algorithm 1.2 adds Selection predicates at many junctures. However, Joins are only added to a translated SQL statement whenever column or table variables appear.

Each Column variable can yield up to a (n-1)-way join of the form “$row_1.Column = row_2.Column \ldots = row_n.Column$”.

Each Table variable can yield up to a (n-1)-way join of the form "$row_1.Table = row_2.Table \ldots = row_n.Table$".

Further, because of the construction, n is at most the number of variables in the input SchemaSQL statement.

The compiled program contains at most one additional join per meta-data variable in the input. Each of these joins is at most a (n-1)-way join where n is the number of variables in the input.

3.3. Lower Bound

If a O(m)+O(p) type bound is accepted by the reader as a trivial lower bound for output size: **i.e.** the compiled program can't be smaller than the input program, and *has* to contain all necessary meta-data since meta-data is not otherwise accessible in SQL, then our compilation can be considered close to optimal by that size measure.

From the point of view of complexity, the number of joins itself produced by the compilation can be seen as a trivial lower bound. Any fewer joins in the output program is impossible since the higher order program contains implicit joins on meta-data variables that appear in the From clause rather than the Where clause. Our compilation inserts an extra join for each meta-data variable entailing such an implicit join. The size and complexity of each of these additional joins is equal to that of the implicit joins they replace (see Section 4.1).

To make this simple 'optimality' argument requires abstracting many details including the optimization of selections and projections. In this argument we only considered join complexity, which we argue is close to a trivial lower bound.

4. Experimental Results

4.1. Comparative Complexity

A simple way to evaluate the performance of the compiler is to compare the execution time of the queries in our running example when they run on Ource (q5, q6), which is in normal form, and when they are expressed on Euter (q1, q2) and Chwab (q3, q4) in their higher order form. Figure 14 shows the computing time for each query against the number of rows in the Euter database. The numbers plotted in Figure 14 show a linear execution time for all queries. However compiled SchemaSQL queries q1, q2, q3 and q4 are slower by a constant multiplicative factor than their SQL counterparts: q5 and q6.

The difference is that on Ource (q5, q6), the queries are expressed with explicit joins using standard SQL. On Euter (q1, q2) and Chwab (q3, q4), the joins are implicit and occur at the interface between meta-data variables and standard SQL variables. This is easily seen in expressions such as "*From DB::T row*". This expression represents an implicit join between meta-data variable *T* and the SQL variable *row*. This implicit join is the result of the inclusion of higher order SchemaSQL algebra operators in query. The compiled queries replace the implicit higher order join, with an explicit SQL join with an intermediate view. This argument is mentioned in the previous section to justify the trivial lower bound on join complexity.

Thus comparing the performance of the queries in our running example can give an indication of how much the compiler costs in terms of complexity when implicit joins are compiled

The database Ource, which is in normal form, contains 50% more cells. The databases Chwab and Euter are more compact since they do not contain the company names in their tables. Measurements are made in seconds on a Postgres database query engine, and stock values are generated according to a progressive random formula

simulating stock-market fluctuations. Static optimizations, described in Section 2 are applied.

4.2. Complexity

An interesting measure of complexity is in terms of the compiler's performance when both the data and the meta-data catalogs grow. Our theorem and theoretical analysis would make us hope that the computation costs grow linearly with the size of the catalog. That theory is comforted by the results graphed in Figures 15 and 16. The catalogs are grown by increasing the number of stocks from 3 to 30. For Euter, the number of attributes grows with the number of stocks, for Chwab the number of tables grows with the number of stocks. Figure 15 groups q1, q3 and q5 together since they all correspond to a three-way join type of operation. Figure 16 groups q2, q4, and q6 together since these queries only represent two-way join. The experimental number show that, *in both cases*, compiled SchemaSQL queries on Euter and Chwab do perform slower than the SQL queries on Ource, but by no more than a constant linear factor. This is best seen in the logarithmic scales of Figures 15 and 16 by noting that the separation between the curves is either constant (Figure 16) or decreases (Figure 15).

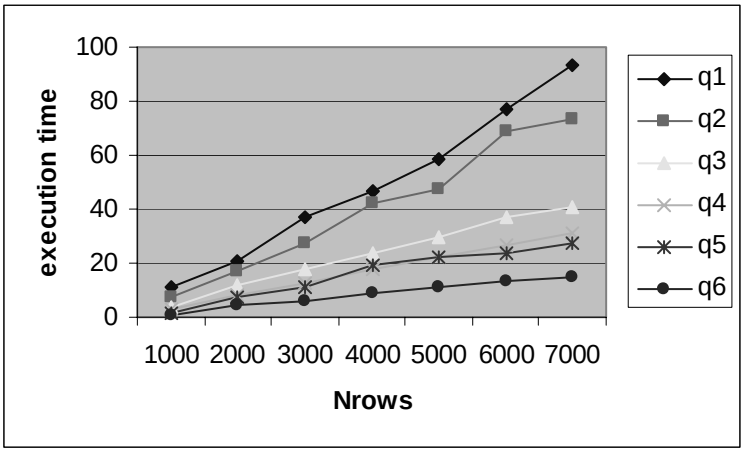


Figure 14 – Execution time vs. number of rows

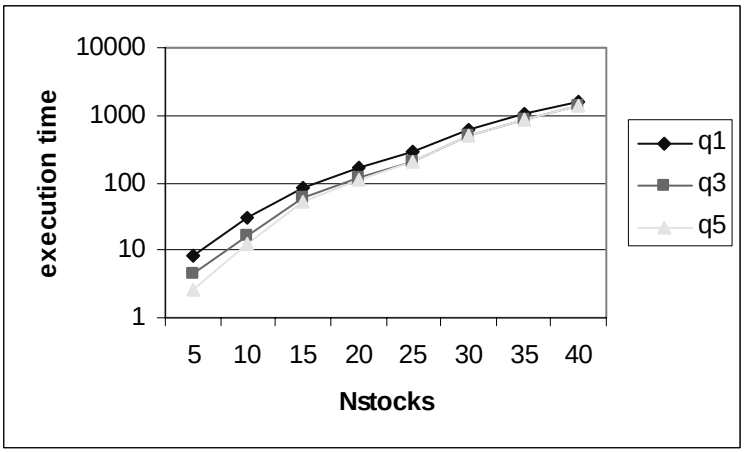


Figure 15 – Execution time vs. catalog size (3 way join)

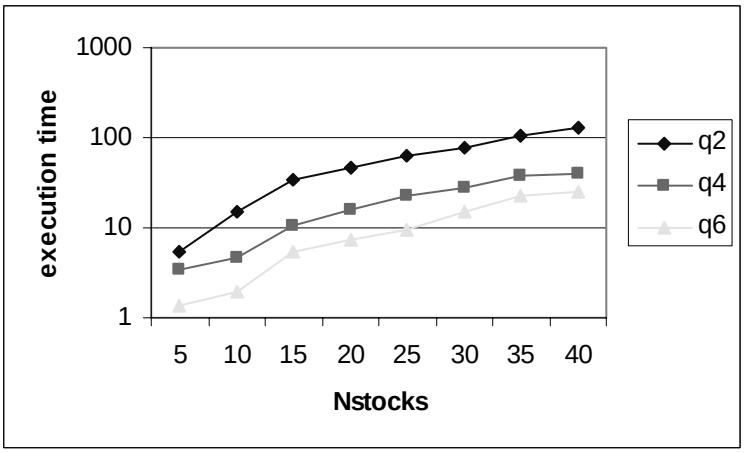


Figure 16 – Execution time vs. catalog size (2 way join)

5. Conclusion

Treating meta-data as data raises the pivotal question of extending a common SQL type system to encompass meta-data. This is a necessary condition of implementing a language such as SchemaSQL as a database solution. But typing is also a way to control complexity of transcribing arbitratry SchemaSQL queries into SQL. Furthermore using the type system properly will allow the SQL engine to optimize SchemaSQL queries in a rational manner. By treating meta-data types as regular tables, each implicit and explicit

second order operator is compiled into a regular join and this process yields a first-order query optimization problem that can be considered by existing optimizers.

Overall this mapping of second order operators into first order joins allows to guarantee that the complexity of the output program is directly correlated to that of the input program, just as the size of the compiled program is also formally bound to the size of original program plus the size of the database's catalog tables.

Last, we suggest that there might be some flexibility in the compilation process. We exhibit at least one such alternative, when it comes to materializing intermediate views. A more sophisticated approach would use our compilation as a starting point and implement a compilation query engine, in which physical plan operators correspond to various compilation methods into regular SQL queries. Such a hybrid system, would preserve the portability of a compilation approach and allow for some flexibility as to how compilation should be performed based on quantitative catalog estimates.

Chapter 5 Resolving Ambiguity through Active Learning

We look at the specific needs of an interactive user interface, and what kind of learning algorithm will be necessary to drive the system. We adopt a version spaces approach, which we adapt to handle a new kind of label. We demonstrate that the modified algorithm conserves the original properties of Version Spaces correctly eliminating candidates inconsistent with labeled examples. We address the problem of selecting examples to drive the induction and propose domain specific heuristics as well as some optimizations designed to improve the combination of learning and sample selection algorithms.

1. Learning Algorithm

We resolve the ambiguity inherent in graphical definition of federating queries by proposing a new learning algorithm. We identify three component sub-problems to the problem of learning federating queries, and we class them into the categories of structure and cardinality. We will introduce two theorems, which form the basis for the formalization of the query discovery problem as a classifier and allow us to further develop the learning algorithm.

1.1. Solution Requirements

- Easy to use: The user does not have to learn any new concepts to communicate with the system.

- Comprehensive. No additional expert help necessary. The user does not have to figure out a way of verifying correctness.
- Efficient enough to complete both in reasonable time and without having to ask the user a large number of questions.

In addition to these requirements other desirable qualities include: minimal user interaction, a meta-model of all possible transformations, and a consistent learner with perfect accuracy and known recall and coverage.

1.2. Approach

One of the conditions for success, is to devise a system that will converge quickly while minimizing the burden on the user. For this purpose we adopt an active learning approach, to minimize the data set [TCM99]. Our plan is to combine this approach with a graphical user interface. A first positive example will be provided by the user, who constructs a row of the federated view by drawing on data from component sources. The system can then switch to active learning, submit rows to the user and ask if they are negative or positive examples. Each submitted row should be chosen to maximize information gain and discriminate between as many possibilities as feasible.

1.3. Learning Queries as a Classifier Problem

1.3.1. Syntactic Structure of Project-Select-Join Queries

As a first approach to the problem, we envision a hypothesis space as a set of second order queries consisting of a single Project-Select-Join (PSJ) clause.

<Query> = Select (<var>

|<var>.<var>

*|<var>.<constant>)**

*From (<var> <RANGE>)**

Where $\wedge$*(<Pred>)**

<RANGE> = (<var>|<Rexp>|<Enum>)	*Range for Variables*
<var>	*Variable*
<Rexp>	*Range Expression*
<Enum>	*Enumeration*
<Pred>	*Predicate Expression*

In the context of first order languages, all variables in the PSJ query would be tuple variables and range over the rows of a table. When the context is second order, variables can range over a set of relations or a set of attributes. This is reflected in the extended syntax, introduced above; the expression RANGE can be formed by a higher order variable.

In the Where line of a PSJ query, the predicates could include any arbitrary expressions using variables defined in the From clause. However of particular interest is the subset of equality predicates that can take two forms:

- Equality with a constant, such as $var_1 == c_0$ or $var_1.var_2 == c_0$, where one side of the equality is a constant.
- Equality as part of a join predicate, such as $var_1 == var_2$, where both sides of the equality contain variables.

1.3.2. Simplifying Assumption

In a first approach, we restrict the subset Θ of PSJ queries that we aim to learn. We leverage the initial user defined graphic example and build a search space containing the queries in Θ that are consistent with it. Our main goal in defining Θ is to adopt restrictions on the language of PSJ queries, such that these restrictions will allow us to initialize a finite search space. Yet Θ will remain expressive and powerful enough to handle all the potential queries that we expect to encounter in a data federating context.

The first restriction for Θ is to limit the Where line in the PSJ clause to equality predicates. The second restriction is to limit the Where line to a conjunction of predicates. Thus, arbitrary arithmetic Boolean formulas will not be explicitly included in our first approach to the learning problem. There will be no self-joins, this means there will be only one variable per table or enumeration. No two variables will range over the same table or enumeration. We also exclude hidden joins from the language defining Θ.

We restrict Θ to single PSJ clauses with no relational union or difference and no aggregation and sorting operators. We also restrict Θ by not allowing embedded clauses. These last two restrictions intend to restrict queries in Θ to a simple PSJ clause, however we can envision forming more complex view definitions by composing and merging several smaller views defined individually. We leave aside for now the possibility of such extensions to Θ.

1.3.3. Query Components

Our representation structures the target concept into three elements:

- Variables and their Ranges (From line)
- Projection onto the Output schema (Select line)
- Join and Selection predicates (Where line)

To know these elements together is to know the target concept. As a triplet, they are exactly equivalent to an instance of the target concept. Each of these elements cannot be represented entirely independently of each other because of the dependencies that exist. However there is a clear division between two major components.

Definition: *Structure Component*

The structural part or *structure component* of the query is formed by the first two elements taken together: the From and the Select lines. Two queries that share the same

structure clauses are both subsets of the same query (see Cardinality component). However they may be disjoint subsets.

Definition: *Cardinality Component*

The *cardinality component* of the query is formed by the Where line. A query with a non-empty cardinality component always yields a subset of the query formed with the same Structure component and an empty Cardinality component. The cardinality component is a filter on the query formed by the Structure component.

1.3.4. Graphic Input Theorem

Our key observation is that the learning algorithm should be a two step process and that each step will correspond to one of the two query components.

The first step will be to learn the structure component of a query. The goal in setting that first step is for the algorithm to leverage the user's GUI interaction to synthesize a structure component. This goal is validated by the Graphic Input Theorem for the learning algorithm. This approach is motivated by our observations when looking at a specific graphic input interface in detail.

As set forth in our taxonomy, a strong example is a set of triplets (table, row, column) of elements from the component databases, such that each triplet is mapped to a column number in the output view. A strong example is the exact representation of the input given to us by the user’s drag and drop actions through our graphic interface.

By contrast a weak example is a set of values (literal value) from the component databases, such that each value is mapped to a column number in the output view. A weak example can be derived from a strong example, by fetching for each triplet the corresponding value from the component databases. Conversely, a weak example cannot derive a strong example, because a data value cannot uniquely pinpoint an origin location in the source databases in terms of (table, row, column): the same literal value will often appear in several locations in the sources.

Theorem 1.3.4 (Graphic Input Theorem). For any simple SchemaSQL clause in Θ with the following restrictions: no hidden joins, no self-joins, no embedded queries, no aggregation or sorting operators; the one meta-data set per element assumption implies that the structure part is uniquely determined by a strong example from the source databases.

1.3.5. Tractability Theorem

The graphic input theorem predicted that the structure component of the target concept will be derived by concentrating on the graphic input to the system. The second step for the learning algorithm is to identify the remaining cardinality component to form the target concept. Our observation is that because of the graphic input theorem, the learning system will only need to search among possible Cardinality components for the query. A set of possible Cardinality components is constructed using the Structure component identified in the first step.

Theorem 1.3.5 (Tractability Theorem). There is a finite set of simple SchemaSQL clauses in Θ that can correspond to a given graphic input.

We have identified a set Θ such that by virtue of some simplifying assumptions, an initial graphic example will specify a unique Structure component, and such that the remaining possibilities for the Where line form a finite space. Θ is the maximal set satisfying both theorems for simple SchemaSQL queries: each of the simplifying assumptions applied to Θ is necessary. We enumerated them as: no hidden or self-joins, no embedded queries, no aggregations. The reader can verify that removing any of them will either yield an infinite search space or violate the graphic input theorem.

The Graphic Input theorem is a convenience but not a necessity for the learning problem. Some of the restrictions to Θ that are introduced by our analysis will not remain inherent limitations of our approach, but simply correspond to the Graphic Input theorem. It is possible to envision learning systems that will lift those restrictions not required by the Tractability theorem. The one meta-data set per element assumption, restrictions on

aggregation and sorting operators, on hidden joins and self-join variables can be relaxed. For example, lifting both of the latter implies a learning system implementing query path joins as demonstrated by the Clio system [YMH01].

1.4. Instantiating a Search Space

We introduce two methods to make the learning of PSJ queries into a classifier problem. The first is a candidate merging method, which takes as input a set of potential view definitions. Each view definition is assumed to be compatible with the same attribute mapping, and could be output by a semi-automated schema-matching tool. Thus we show how to build a search space by merging different queries, but still assuming that the schema matching problem has already been resolved. Next we will introduce a method, which makes no such assumptions, and will build a default search space, using only an initial data mapping provided by the user.

1.4.1. Search Space Initialization by Query Merging

Consider an initial set of view definitions $VD_1, VD_2, \ldots, VD_n$. We write each of those view definitions VD_i as an algebra sentence using projections, selections and a Cartesian product CP_i. We assume that all those view definitions are compatible with a common attribute mapping, such as the one shown in Figure 18. This means that all agree on which source element to map to each column in the target table. In practice this implies that each of the view definitions can be written with the same formal projection operator to the target view: $\Pi(a_1, \ldots, a_t)$, where $a_1, \ldots, a_t$ are attributes from source tables.

The correct semantic to describe Cartesian products CP_i is not a set of relations, but rather a bag. Certain relations may appear more than once in a Cartesian product, $CP_1, \ldots, CP_n$. For example, if two mapped attributes originate from the same relation, one view definition could feature a self-join of that relation, while another does not. By using the bag maximum operation defined by Kent ([Kent92]), each view definition can be rewritten using the maximum Cartesian product $CP_{max} = max_{bag}(CP_1, \ldots, CP_n)$ of all Cartesian products. Assuming that each table features a primary key, this rewriting can

done transparently by adding join predicates in the view definitions, in order to maintain the correct join semantics. For example, if the same relation O_1 appears twice in CP_{max}, but self-join is not the desired semantic for some view definition VD_i, it is possible to add a join predicate to VD_i, between the instances of that duplicated relation. Formally if $O_1=O_1'$ then $\Pi_{(...)}(O_1) = \Pi_{(...)}(\sigma_{O1.pk=O1'.pk}(O_1 x O_1'))$.

$VD_1 = \Pi(a_1, \ldots, a_t)$
$\quad (\sigma_{1,1} \circ \sigma_{1,2} \circ \ldots \circ \sigma_{1, \lambda(1)}$
$\quad\quad (O_1 x O_2 x \ldots x O_c))$
$VD_2 = \Pi(a_1, \ldots, a_t)$
$\quad (\sigma_{2,1} \circ \sigma_{2,2} \circ \ldots \circ \sigma_{2, \lambda(2)}$
$\quad\quad (O_1 x O_2 x \ldots x O_c))$
$\ldots$
$VD_n = \Pi(a_1, \ldots, a_t)$
$\quad (\sigma_{n,1} \circ \sigma_{n,2} \circ \ldots \circ \sigma_{n, \lambda(n)}$
$\quad\quad (O_1 x O_2 x \ldots x O_c))$

(a) – Alternative View Definitions under Consideration

$VD(F_1, F_2, \ldots, F_{pf}) =$
$\Pi(a_1, \ldots, a_t)$
$\quad (F_1 \sigma_{1,1} \circ F_2 \sigma_{1,2} \circ \ldots \circ F_{\lambda(1)} \sigma_{n, \lambda(n)}$
$\quad\quad F_{\lambda(1)+1} \sigma_{2,1} \circ F_{\lambda(1)+2} \sigma_{2,2} \circ \ldots \circ F_{\lambda(1)+\lambda(2)} \sigma_{2, \lambda(2)}$
$\quad\quad \ldots$
$\quad\quad F_{\Sigma\lambda(i)+1} \sigma_{n,1} \circ F_{\Sigma\lambda(i)+2} \sigma_{n,2} \ldots F_{pf} \sigma_{n, \lambda(n)}$
$\quad\quad (O_1 x O_2 x \ldots x O_c))$

(b) –View Definition Formula for Boolean Vector ($F_1, F_2, \ldots, F_{pf}$)

$VD_1 = VD(F_1=True, F_2=True, \ldots, F_{\lambda(1)}=True, False, False, \ldots, False)$
$VD_2 = VD(False, \ldots, False, F_{\lambda(1)+1}=True, \ldots, F_{\lambda(1)+\lambda(2)}=True, False, \ldots, False)$
$\ldots$
$VD_n = VD(False, \ldots, False, F_{\Sigma\lambda(i)+1}=True, \ldots, F_{\Sigma\lambda(i)}=True)$

(c) – The original View Definitions are still expressible using the parameterized Formula.

Figures 17 (a), (b), (c) – Search Space Initialization by Query Merging

Thus, assuming that all alternative view definitions under consideration are compatible with the same attribute mapping, we can require that they be rewritten to use the same Cartesian product, and the same projection formula. They will only differ in their selection and join predicates (Figure 17a). We can then merge those view definitions into a formula parameterized by Boolean variables $F_1, F_2, \ldots, F_{pf}$ (Figure 17b). Each of those Boolean variables F_i is combined with a selection operator σ_i, such that if F_i is false the selection operator σ_i becomes neutral (the identity function) and if F_i is true the

selection operator σ_i applies normally. Each of the original view definitions in the merger can still be expressed within the parameters of this Boolean formula (Figure 17c).

The resulting formula: $VD(F_1, \ldots, F_{pf})$ represents the search space of possible view definitions. The learning algorithm will seek to converge by finding the correct assignments to each Boolean variable or feature in $F_1, F_2, \ldots, F_{pf}$.

Unlike the default approach, this method will work even if the queries to be merged contain disjunction, as long as they are expressed in CNF form. Neither do the predicates need to be restricted to simple equalities. However, our method does not handle disjunction as a general case, neither in the Where line, nor by union of several PSJ clauses: we seek to combine the original view definitions on a purely conjunctive basis.

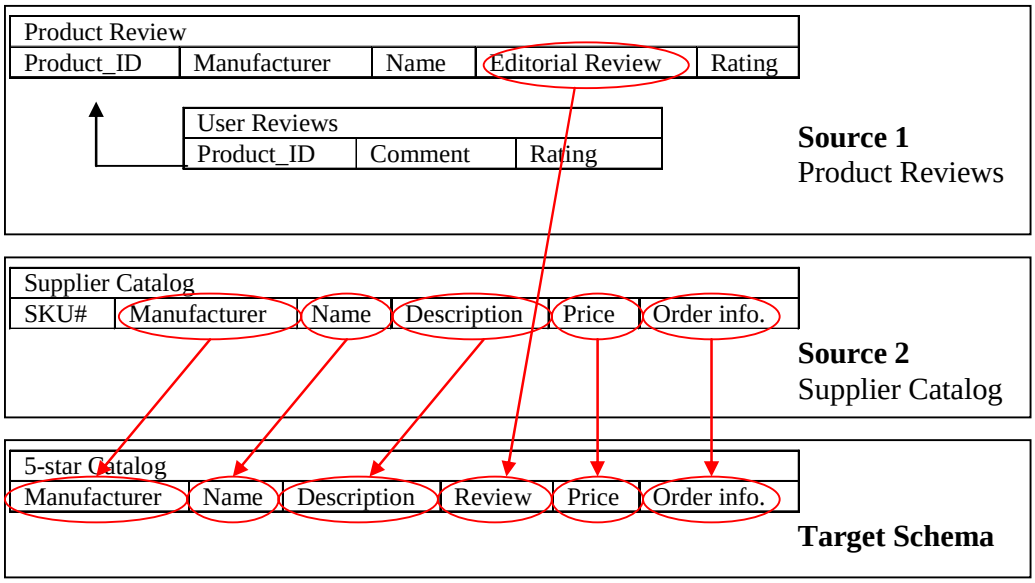


Figure 18 – Federating Data Sources

1.4.2. Default Search Space Initialization

A search space can also be initialized using an initial data mapping as shown in Figure 19. Such an initial data-mapping example can be generated by a user with a QBE-like, point and click interface. Constructing such an example does not require peculiar

abstraction skills from the user, however it is assumed that as a domain expert, the user knows and understands the data sufficiently to produce a correct mapping.

Note that a data-mapping example is richer than a simple attribute mapping: compare Figures 18 and 19. The *data* mapping example immediately gives us the proper mapping of attributes from source to target also found in the *schema* mapping, but also contains more information because it is grounded with actual data instances from each source. A data mapping as shown in Figure 19 is a strong example, and in accordance with the Graphic Input theorem, will enable us to initialize a search space for query discovery.

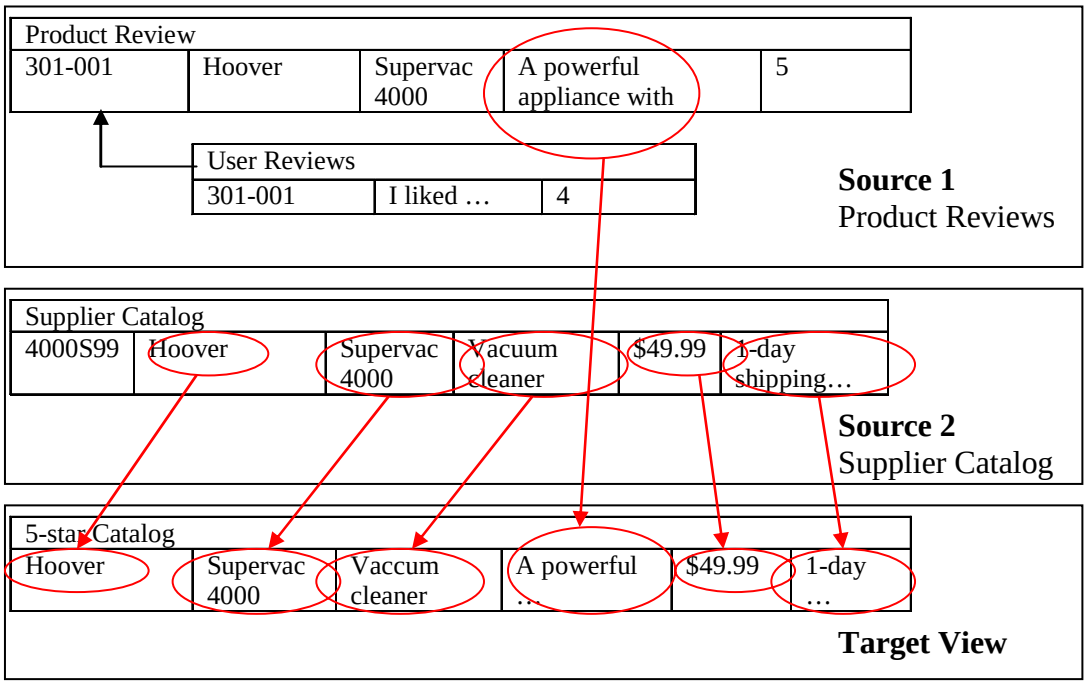


Figure 19 – Data Mapping Example

With the following procedure, we define a parameterized Boolean formula, which will represent the initialized search space:

$$VD(F_1,\ldots, F_{pf}) = \Pi(E_1, \ldots, E_t)\ (\sigma_1\ o \ldots\ o\ \sigma_k\ (\ O_1\ x\ O_2\ x \ldots O_c))$$

***Procedure* 1.4.2 (Search Space Initialization)**

Step 1.

- Introduce a Cartesian product between every tables, which appears in the data mapping example: O1xO2x…xOc.

For each table O_i, introduce variable $\$r_i$.

Step 2.

- Introduce the projection operator necessary to form the target view from the Cartesian product: $\Pi(E_1, \ldots, E_t)$, where each E_i is a relational expression using $\$r_1, \ldots, \$r_c$.

Form the Select clause

Step 3.

- Create equality selection predicates $\sigma_1, \sigma_2, \ldots, \sigma_k$ for every value, which is part of a row used in the data mapping.

For each table Oi, where o_i is the row from O_i used in the data mapping, introduce for every attribute a_j of O_i, predicate σ: "$\$r_i.a_j = o_i.a_j$"

Step 4.

- Create equality join predicates $\sigma_{k+1}, \sigma_{k+2}, \ldots, \sigma_{pf}$ for every pair of values in the data mapping which are in distinct tables and are equal.

For every pair o_i, $o_{i'}$ of mapped rows such that for some pair of attributes a_j, $a_{j'}$: $o_i.a_j = o_{i'}.a_{j'}$, introduce predicate σ: "$\$r_i.a_j = \$r_{i'}.a_{j'}$".

We apply this procedure to the example illustrated in Figure 19:

- The Cartesian product is **Product Reviews** x **Supplier Catalog**. The corresponding row variables are PR and SC.
- It follows that the projection operator is $\Pi_{(SC.Manufacturer, SC.Name, SC.Description, PR.Ed.Review, SC.Price, SC.Orderinfo.)}$
- The selection predicates are:

σ_1: "PR.Product_ID='301-001'",

σ_2: "PR.Manufacturer='Hoover'",

σ_3: *"PR.Name='Supervac 4000'"*,

σ_4: *"PR.Ed.Review='A powerful…'"*,

σ_5: *"PR.Rating='5'"*,

σ_6: *"SC.SKU='301-001'"*,

σ_7: *"SC.Manufacturer='Hoover'"*,

σ_8: *"SC.Name='Supervac 4000'"*,

σ_9: *"SC.Description='Vacuum …'"*,

σ_{10}: *"SC.Price='$49.99'"*,

σ_{11}: *"SC.Orderinfo='1-day shipping'"*.

Note that the total number of selection predicates is equal to the total number of attributes of tables **Product Reviews** and **Supplier Catalog.**

- The join predicates are

σ_{12}: *"PR.Manufacturer = SC.Manufacturer"*

and σ_{13}: *"PR.Name=SC.Name"*.

Note that the data mapping example given by the user immediately excludes a join on attributes *Product_ID* and *SKU#*: the corresponding join predicate is not even generated.

This default search space, while fairly general is far from complete. For example, it does not generate view definitions in which the same table appears more than once (self-joins) and only equality predicates are generated. The search space does correspond precisely to the syntactic elements of the language used to express the federating view definitions. The power of the system can be expanded by carefully expanding the syntax and expressiveness of the federating view language and concomitantly adding features to the search space. For example, adding outer joins to the search space would require

generating an Outer-Join predicate *(t1.fk = t2.pk OR (t1.fk = null AND t2.pk = blank) OR (t2.pk = null AND t1.pk = blank))* for each potential join key *[tab1.fk ↔ tab2.pk]*.

It is possible to make the default search space more complete, by adding extra features such as outer joins, and joins along all possible query paths, but these come at the expense of extra features, and still do not guarantee completeness. The size of the search space doubles with each new feature. But while the search space grows exponentially with the expressiveness of the language, our meta-model representation of the search space can remain linear ([Hirsh92], [HMP97]), and the time required to converge can be controlled by heuristics.

The current language is expressive enough to be widely effective and represent a large class of challenging queries. Exploration of a language of commercial interest is beyond the scope of this paper.

1.5. Learning Queries

We seek to define a target concept (what the system is trying to learn) for the context of learning federating view definitions. If the target concept is a set, "Positive examples" are objects that belong to the target concept, and "Negative examples" objects that do not belong. The system has successfully "learned" the target concept when it can correctly identify whether any given objects is in the target concept or outside of it.

Given the problem definition of learning how to build a given federated view, the target concept in a learning sense could be a Table. However a desirable target concept could also be the specification of the federated view as a second order or SchemaSQL query. These two notions are not computationally equivalent, and we adopt the terminology of *target query* and *target view* to identify these distinct goal concepts for the learning algorithm.

Definition: *Target View, Target Query*

The *target view* is the materialized view that the learning system (Sphinx) is trying to learn from the user. The *target view* is defined by the execution of the *target query* over the source databases.

We observe that since the learning algorithm will focus on active learning techniques to solicit feedback from the user on additional examples. The goal is for this feedback to discriminate precisely among the set of possible Cardinality components and allow the algorithm to identify a unique solution. This set of Cardinality components or potential predicates maps exactly to the feature vector of a classifier if the chosen target concept is the Target View.

However, our real goal is for Sphinx to come up with a correct view definition, i.e. learn the Target Query rather than the Target View. It will naturally occur to the reader that over a given data source, there may be several queries, which yield the same materialized view as the target query:

Consider the equation TV = TQ(DB), the target view TV is the product of the databases DB transformed by the target query TQ. This equation, given TV and DB has more than one solution in TQ. As such Sphinx cannot always converge on the correct target query. However, Sphinx will identify the entire class of queries, which correctly materialize the target view over the source data and picks the appropriate one.

The set of rows DM that are produced by the structure part of the query is divided into two sets. A positive set DM^+ contains rows that successfully pass the cardinality criteria. In that context we consider that positive examples are rows that belong to the target view, and that negative examples are rows that do not belong. Positive examples come naturally from user interaction. Like in any learning system, both negative and positive examples must form the training set, and we will need negative examples.

Versions Spaces fits our problem definition. It allows us to keep track of the shrinking hypothesis space as the interaction progresses and more examples are introduced to the system. Version Spaces introduced by Mitchell can easily learn the characteristic features of a concept, based on a conjunctive description ([Mitchell77]).

The learning is inductive and proceeds by giving the algorithm positive and negative instances. In this section, we will assume that our goal is to adapt Version Spaces to databases, using a view as the concept, and individual database rows as either positive (the row is in the view) or negative examples (the row is not in the view). The difficulty in adapting Version Spaces to our problem is twofold. First, not all examples can be classified as either positive or negative. Rows with certain combinations of features will violate certain database dependencies or implicit 'domain' related constraints, thus their viability as examples presented to the user for labeling is dubious. Further, rows with certain combinations of features will be theoretically possible, but completely imaginary, making it impossible for our system to present to the user an example, which is grounded in actual data. It will be up to our system to determine, that indeed through dependency constraints or through insufficient data, such is the case for certain combination of features. Ultimately, one may argue that under certain circumstances such cases are more akin to negative examples than positive examples. This will be the basis for extending the Version Spaces learning algorithm. Second, an active learning shell needs to be added to the Version Spaces process in order to allow the selection of examples to submit to the user. Our algorithm is adapted from Version Spaces to detect and accommodate the case of missing examples, and to preserve useful correctness properties for the final view definition resulting in such a database context.

1.6. Boolean Feature Vector

Either initialization procedure assembles a group of selection and join predicates. The total number of potential predicates being considered is *pf*. This predicate set: $\{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_{pf}\}$ determines a search space of federated view definitions, in which both the Select and From clauses are invariants:

- The set of tables $O_1, \ldots, O_c$ in the cartesian product determines the From clause as "*From* O_1 *\$r*$_1$, O_2 *\$r*$_2$, …, O_c *\$r*$_c$"
- The projection expression $\Pi(E_1, \ldots, E_t)$ determines the Select clause as "*Select* E_1, E_2, …, E_t"

The Where line in our search space consists of a conjunction of predicates, taken from the set $\{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_{pf}\}$. Hence, the semantics of a federated view definition in our search space will be determined solely by which subset of the set of potential predicates, appears in the Where clause. We introduce the notion of *feature vector* as a Boolean vector of size pf. A *query feature vector* is the practical and unambiguous specification of a query in our search space.

Definition: *Query Feature Vector*

The *query feature vector* associated with a query q in the search space, and abbreviated *FV(q)* is a Boolean vector of size pf. Given fixed *Select* and *From* clauses for the search space, if $FV(q) = (q_1, \ldots, q_{pf})$, q is the query, and such that $q_i=1$ if and only if filter predicate σ_i appears in the *Where* clause of q.

There are exactly 2^{pf} queries in the Search space, 2^{pf} distinct feature vectors, and we have defined a one-to-one mapping between queries in that space and their query feature vectors. In the rest of this text, we will limit our discussion exclusively to those queries which are in this Search Space or Hypothesis Space, such that for each query q there is a corresponding *query feature vector* FV(q).

Definition: *More Specific Than, More General Than*

Let *a* and *b* be two feature vectors such that $a = (a_1, \ldots, a_{pf})$ and $b = (b_1, \ldots, b_{pf})$.
a is *more specific than* b (noted $a \geq b$) if and only if $(\forall i: a_i \geq b_i)$,
b is *more general than* a if and only if a is more specific than b.

Figure 20 illustrates the overall question/answer approach to solving the query discovery problem. Note that this whole mechanism is hidden from the user, whose

interaction is limited to answering yes or no.

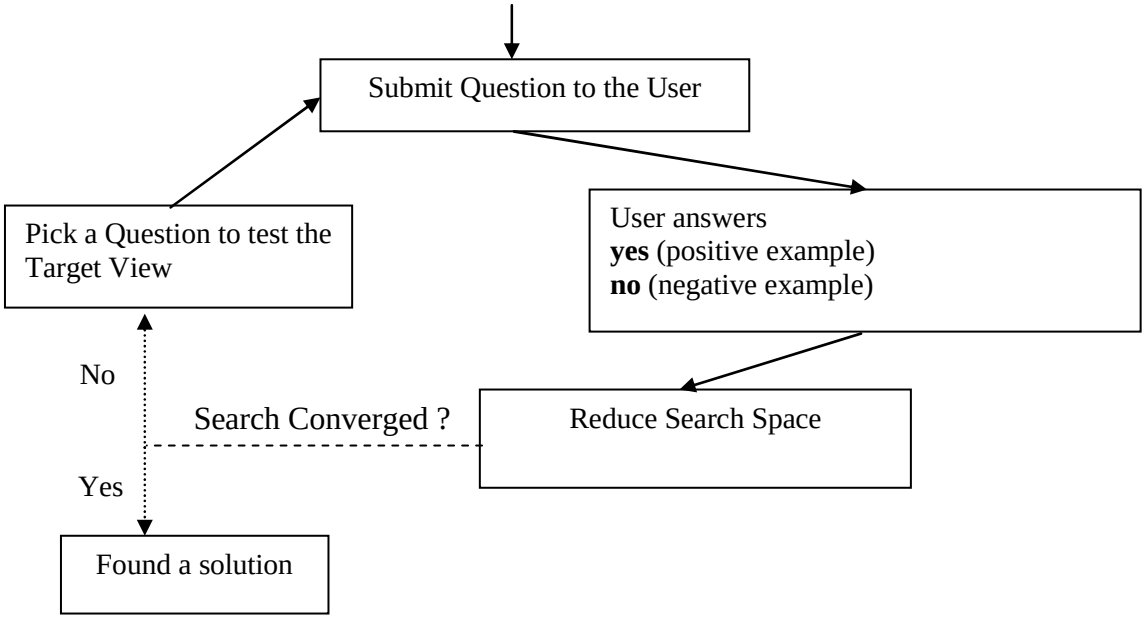


Figure 20 – Question/Answer Interaction

The Sphinx learning algorithm performs an iterative loop, whose termination condition is the convergence of the Version Spaces. At each step a feature vector is chosen, and labeled with one of three labels: *positive*, *negative* or *missing*. The missing label is the new label we introduce to extend Version Spaces. As a result of these repeated steps the target query can be narrowed to a dwindling subset of candidates by the application of three elimination rules. The narrowing subset is commonly called the *space of remaining hypothesis*. The learning algorithm *converges* when that space is reduced to a single query.

1.7. Version Spaces State

We introduce the concept of Version Spaces state for the purpose of tracking the space of remaining hypothesis.

Definition: *Version Spaces State*

A *Version Spaces state* is a pair of items ***(s, G)*** such that:

- *s* is a query feature vector called the most specific feature vector

- *G* is a set of query feature vectors called the most general set.

As shown in Figure 21, the Version Spaces is initialized with the initial state *(s_0, G_0)*, with $s_0 = (1, 1, \ldots, 1)$ and $G_0 = \{ (0, 0, \ldots, 0) \}$.

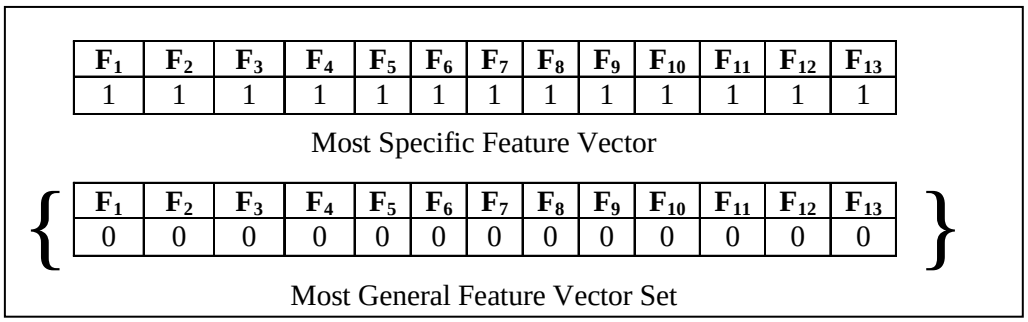


Figure 21 – Initial Version Spaces state

We define the notion of *query set* for a given Version Spaces state as the space of remaining hypothesis. The initial query set $QS(s_0, G_0)$ is equivalent to the entire search space.

Definition: *Query Set*

Let (**s**, **G**) be a pair of items such that the first item **s**, is a query feature vector and the second item **G** is a set of feature vectors: $\mathbf{G} = \{g_0, g_1, \ldots, g_{mg}\}$. The *Query set* for (**s**, **G**) is noted QS(**s**, **G**) such that:

$QS(\mathbf{s}, \mathbf{G}) = \{q \mid q \text{ is more general than } s \text{ and } \exists g_j \in G \text{ such that } g_j \text{ is more general than } q\}$.

1.8. Data Mapping

We defined the notion of data mapping instance to formalize the concept of source data coming together to form an object in the target schema. A data mapping is a positive example if the given source data correctly forms a member of the target view.

Definition: *Data Mapping*

A *data mapping* instance dm is an assignment $(o_1, o_2, \ldots, o_n)$ of variables $\$r_1, \$r_2, \ldots, \$r_n$ in their respective object sets $O_1, O_2, \ldots, O_n$: dm is the n-tuplet $(o_1, o_2, \ldots, o_n) \in O_1 \times O_2 \times \ldots \times O_n$

Definition: *Positive, Negative Data Mapping Example*

A *data mapping* instance dm = (o_1, o_2, …, o_n) is said to form a *positive example* if and only if the target schema object formed by $\Pi_{E1, ..., Ek}$ (o_1, o_2, … , o_n) is a member of the target view. If not, dm is said to form a *negative example*.

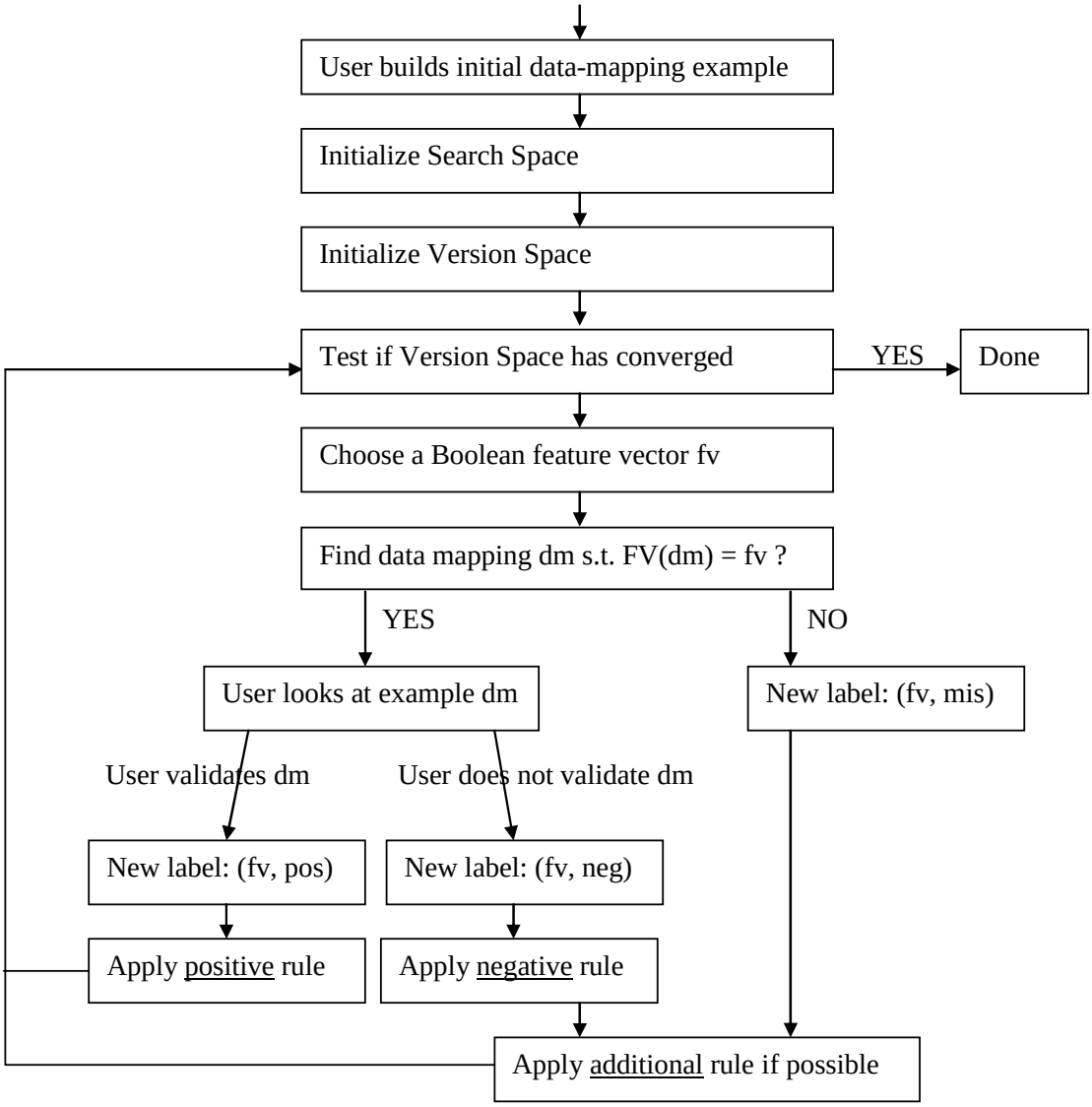


Figure 22 – Chain of Events in the Sphinx active learning algorithm

Figures 19, 23, 25 and 30 illustrate the graphical representation for a data mapping. A data mapping can always be represented by showing a set of the data values in the source databases, combining to form an element in the target schema. If the element

produced is a member of the target view, the data mapping represents a positive example (Figures 19, 23 and 30), if not it represents a negative example (Figure 25).

Just as we did for queries, we can associate a Boolean feature vector with each data mapping instance dm.

Definition: *Example Feature Vector*

For a given data mapping dm = (o_1, o_2, …, o_n), the *example feature vector* FV(dm) is defined as FV(dm) = ($\sigma_1(o_1, \ldots, o_n)$, $\sigma_2(o_1, \ldots, o_n)$, …, $\sigma_{pf}(o_1, \ldots, o_n)$).

1.9. Rule Definition

We introduce the concept of rules, and the three kinds of rules used in the Sphinx learning algorithm. Formally we define a rule as one of three operators acting on a Version Spaces state. In turn, we will formally define three rules.

Definition: *Rule*

A *rule* is an operator, which takes a Version Spaces state *S = (s, G)* and a feature vector *fv* as input and returns a new Version Spaces state *S' = (s', G')*.

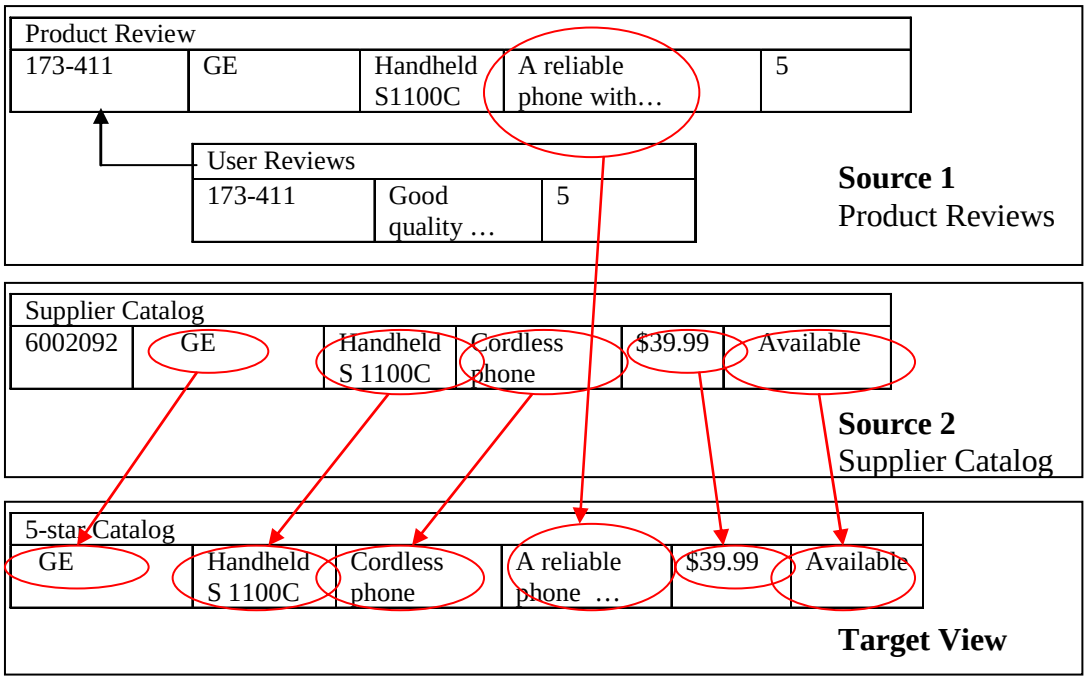


Figure 23 – Positive Data Mapping Example dm_1

1.9.1. Positive Rule

The positive rule operator R_p modifies the Version Spaces state by eliminating from the query set those queries that are inconsistent with a given positive data mapping.

Definition: *Positive Rule Operator*

Let dm be a data mapping such that $FV(dm) = (e_1, \ldots, e_{pf})$. The positive rule operator $R_p((e_1, \ldots, e_{pf}))$ operates on a Version Spaces state $(s = (s_1,\ldots, s_{pf}), G = \{g_1, .., g_n\})$, and $g_i = (g_{i,1} , g_{i,2} , \ldots, g_{i,pf})$.

$R_p((e_1, \ldots, e_{pf}))$: $(s,G) \rightarrow (s',G')$

with $s' = (s_1', \ldots, s_{pf}')$ such that:

- $e_i = 1 \Rightarrow s_i' = s_i$
- $e_i = 0 \Rightarrow s_i' = 0$

and $G' = G$

Consider the positive data mapping instance dm_1 shown in Figure 23. The mapped values are circled. Because the data in dm_1 is substantially different from the initial data-mapping example in Figure 19, a large number of the potential features predicates do not hold on dm_1: $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_6, \sigma_7, \sigma_8, \sigma_9, \sigma_{10}, \sigma_{11}\}$. Conversely the following predicates do hold on dm_1: $\{\sigma_5, \sigma_{12}, \sigma_{13}\}$. Thus the example feature vector for dm_1 is: $FV(dm_1) = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1)$. The result of the subsequent application of $R_p(FV(dm_1))$ to the initial Version Spaces state (Figure 21) is shown in Figure 24. Only the most specific vector is modified: all potential features which are negated in dm_1 see their vector value lowered from 1 to 0. Potential features whose predicates are still fulfilled $\{\sigma_5, \sigma_{12}, \sigma_{13}\}$ suffer no change.

The changes shown in Figure 24 eliminate from the query set, all the queries with any of the negated predicates $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6, \sigma_7, \sigma_9, \sigma_{10}, \sigma_{11}, \sigma_{12}, \sigma_{13}, \sigma_{14}\}$ in their *Where* clause: since those queries would not produce a target view where dm_1 could be a positive example. This property is formally stated in Lemma 1.

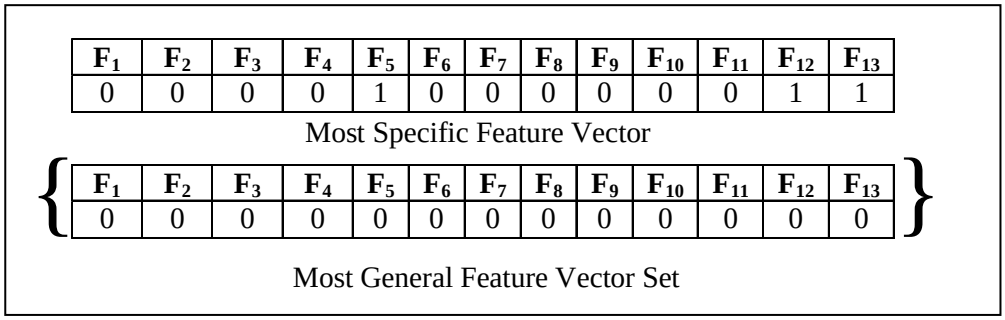


Figure 24 – Applying the *positive rule* operator $R_p(FV(dm_1))$ to the initial Version Spaces state.

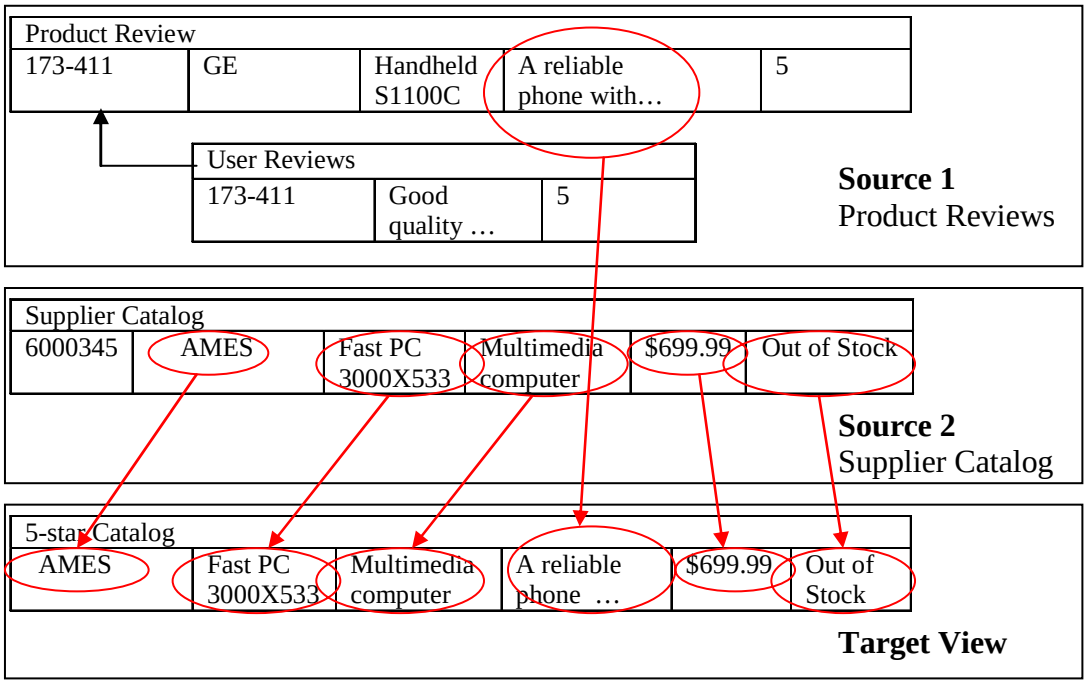


Figure 25 – Negative Data Mapping Example dm_2

1.9.2. Negative Rule

The negative rule operator R_n modifies the Version Spaces state in order to eliminates from the query set those queries that are inconsistent with a given negative data mapping.

Definition: *Negative Rule Operator*

Let dm be a data mapping, such that $FV(dm) = (e_1, \ldots, e_{pf})$.

The negative rule operator $R_n((e_1, \ldots, e_{pf}))$ operates on a Version Spaces state $(s = (s_1,\ldots, s_{pf}), G = \{g_1, .., g_n\})$, with $g_i = (g_{i,1} , g_{i,2} , \ldots, g_{i,pf})$.

$R_n((e_1, \ldots, e_{pf}))$: (s,G) → (s'',G'')

with s'' = s

and G' =

$$\{ (g_{i,1}' , \ldots, g_{i,pf}') \mid [(\exists p: (g_{p,1} ,\ldots, g_{p,pf}) \in G) \wedge (\forall j: e_j=0 \Rightarrow g_{p,j} = 0))$$
$$\wedge (\exists f: (\forall j \neq f: g_{i,j}' = g_{p,j}) \wedge e_f=0 \wedge g_{i,f}' = 1)]$$
$$\vee [(\exists p: (g_{p,1} ,\ldots, g_{p,pf}) \in G) \wedge (\exists f: e_f=0 \wedge g_{p,f} = 1) \wedge (\forall j: g_{p,j} = g_{i,j}')] \}$$

and G'' =

$$\{ (g_{i,1}' , \ldots, g_{i,pf}') \mid (g_{i,1}' , \ldots, g_{i,pf}') \in G' \wedge (g_{i,1}' , \ldots, g_{i,pf}') \leq s'\}$$

Consider the data mapping dm_2 shown in Figure 25. It differs slightly from dm_1, in that the data for source 1 is the same (the product is a GE handheld S1100C), but it is matched with a totally different product from source 2. The feature vector for dm_2 is $FV(dm_2) = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$. There are 12 potential features whose predicates are negated in dm_2: $(F_1, F_2, F_3, F_4, F_6, F_7, F_8, F_9, F_{10}, F_{11}, F_{12}, F_{13})$.

The application of the $R_n(FV(dm_2))$ operator leaves s, the most specific vector unchanged and applies only to G, which has only one member at this stage: $G = \{g_0\}$. The application of $R_n(FV(dm_2))$ happens in two phases:

- In the first phase a new vector g_i' is created from g_0 by changing the bit of exactly one of the 12 features to 1. Since there are 11 features there will be 11 new vectors: $g_1' = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$, $g_2' = (0, 1, 0, 0\ldots., 0)$ up to $g_{12}' = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)$.
- In the second phase only those vectors, which are still more general than s will be kept. This eliminates all but g_{11}' and g_{12}', thus $g_1'' = g_{11}'$ and $g_2'' = g_{12}'$. The final

result is shown in Figure 26.

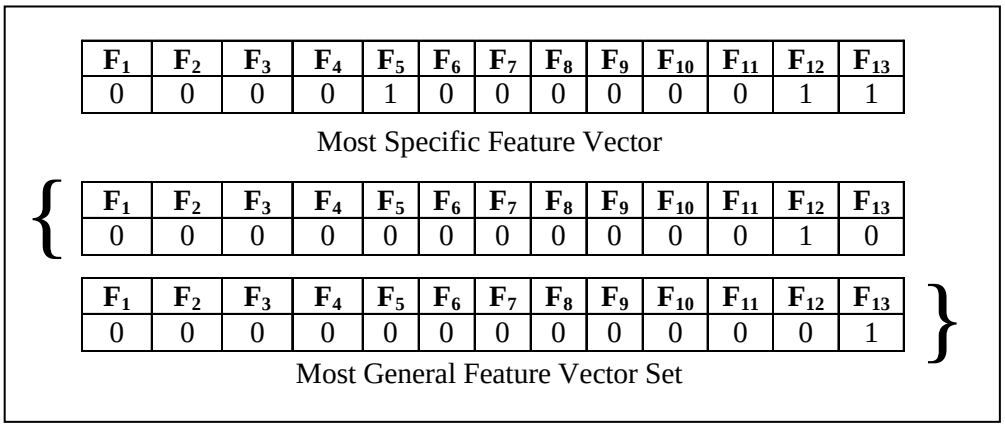


Figure 26 – Applying the *negative rule operator* $R_n(FV(dm_2))$ to the Version Spaces state

1.9.3. Additional Rule

The additional convergence rule operator $R_a(p)$ is applied in the negative and missing label branches every time its precondition is met for some potential feature p, $1 \leq p \leq pf$. If the precondition is not met for any p, the Version Spaces state is unchanged and nothing further happens. If it is met for some p, then the operator $R_a(p)$ is applied and the Version Spaces state is modified to reflect the information derived from the precondition being fulfilled.

Definition: *Set* 0_p

The *set* 0_p is the set of feature vectors containing a zero at the p^{th} position.

$$0_p = \{ (e_1, e_2, \ldots, e_{pf}) \mid e_p = 0\}$$

Definition: *Precondition Ra(p)*

Precondition(p) is met, if for any data mapping *dm*, $FV(dm) \in 0_p$ implies that dm is a negative example. If there is no data mapping dm, such that $FV(dm) \in 0_p$, then *Precondition(p)* is true.

$$(\forall dm : FV(dm) \in 0_p \Rightarrow dm \text{ is a negative example}) \Rightarrow Precondition(p) \text{ is true}$$

Definition: *Additional Rule Operator*

This operator can only be applied when when Precondition(p) is true

$R_a(p)$: (s, G) $\rightarrow$ (s', G')

with G' = G

and s' =

$(s_1', \ldots, s_{pf}'), s_p' = 1 \wedge (\forall i \neq p : s_i' = s_i)$

The original Version Spaces algorithm did not require such a rule. The necessity for the *additional rule* is dictated by the demands of a sample selection system. There will be cases when no data mapping instance for a given feature vector can be found from the existing data sources. Consider predicate σ_5*: PR.Rating = "5-star"*. Assume all objects in the source have the rating "5-star". Thus all positive data mapping instances validated by the user must contain the rating "5-star". This is insufficient to prove that predicate σ_5 is part of the target view since all negative data mappings must also contain the "5-star" rating. Disproving the presence of σ_5 in the *Where* clause is similarly impossible. Thus, as no examples can be found with a different rating, the system will never be able to determine if the predicate σ_5 should appear or not in the *Where* clause of the target concept.

It should be noted that since all data instances in the source fulfill predicate σ_5, its presence in the target query does affect the content of the target view. Every query *q* containing σ_5 in the *Where* clause, has an identical counterpart *q'* which is similar to *q*, except for not having σ_5 in the *Where* clause. It is always the case that *q* and *q'* materialize the same target view. In that case, applying the *additional rule operator* $R_a(5)$ will reduce the query set by removing for every query *q*, its counterpart *q'*.

1.9.4. Impact on Updates

Consider the scenario where the additional rule operator $R_a(5)$ is applied because no data mapping can be found with a rating other than "5-star". Assume that as the result of an update or a modification of the source data, a rating of "4-star" appears at some point in the future. It will be necessary for Sphinx to re-evaluate the target concept against this new information. Thus every application of the *additional rule operator*

should be logged, and a trigger should be introduced as an integrity constraint on the source data.

For each application of an additional rule operator $R_a(p)$, *Precondition(p)* must become an integrity constraint on the source data. Such an integrity constraint is violated when a data mapping *dm* appears in the source such that:

- $FV(dm) \in 0_p$
- dm has not been labeled a negative data mapping.

Violation of this integrity constraint, and of *Precondition(p)* must trigger a restart of the learning algorithm at the point where the additional rule operator $R_a(p)$ was applied.

2. Correctness

In this section we look at correctness and prove that the Sphinx learning algorithm, if it converges, will identify the correct target concept in the search space. To do so, we prove that each rule operator application removes from the space of remaining hypothesis only those queries, which are inconsistent with the data mapping labels. We also prove that the learning algorithm terminates by exhausting the search space after a finite number of steps.

2.1. Data Mapping and View Mapping Set

The *data mapping* set corresponds to the Cartesian product of the Cartesian sets $O_1, O_2, \ldots, O_n$ defined by either initialization method. We also define the *positive data mapping* set as a subset of the *data mapping* set.

Definition: *Data Mapping Set DM*

The set DM of data mapping instances is formally defined as the Cartesian product of the object sets $O_1, O_2, \ldots O_n$.

Definition: *Positive Data Mapping Set DM+*

Given a query *q*, such that $FV(q) = (q_1,\ldots, q_{pf})$, the positive data mapping set $DM^+(q)$ is the subset of *DM* which does not negate any of the predicates included in the query *q*.

Note that if q_0 is the query with feature vector $FV(q_0) = (0, 0, \ldots,0)$, then $DM^+(q_0) = DM$. Note that if *a* and *b* are two queries such that *a* is more specific than *b* then $DM^+(a) \subseteq DM^+(b)$.

Data Mapping Set:

$$DM = O_1 \times O_2 \times \ldots \times O_n$$

Positive Data Mapping set for query q:

$$DM^+(q) = \{ (o_1, \ldots, o_n) \in DM \mid \forall i : q_i \Rightarrow P_i(\$r_1=o_1, \ldots, \$r_n=o_n) \}$$

View defined by query q:

$$View(q) = \{ \prod(E_1(\$r_1=o_1, \ldots, \$r_n=o_n), \ldots, E_k(\$r_1=o_1, \ldots, \$r_n=o_n)) \mid (o_1, \ldots, o_n) \in DM^+(q) \}$$

$$View(q) = \Pi_{(E1, \ldots, Ek)}(DM^+(q)) = \Pi_{(E1, \ldots, Ek)} (\sigma_{1 \ldots} \sigma_m (DM))$$

Figure 27 – Sets defined by a query

2.2. View Defined by a Query

Any query q in the search space defines a view over the set of source databases. Two different queries q_1 and q_2 may define the same view. A trivial illustration of this is when the source databases are empty.

Definition: *View Defined by a Query q*

Given a query q, such that $FV(q) = (q_1, \ldots, q_{pf})$, we note View(q) the view defined by q.

In a machine learning view of the query discovery problem for target query TQ, elements of DM serve as labeled examples: positive examples belong to $DM^+(TQ)$, and negative examples do not belong to $DM^+(TQ)$.

Lemma 1 states that a data mapping dm is a positive example for query q if and only if feature vector FV(dm) is more specific than feature vector FV(q). Lemma 1 states the feature vector properties of the elements of $DM^+(q)$ for queries q in the hypothesis space. Lemma 1 allows us to consider labels for positive or negative examples as a property of feature vectors, by stating that the label of a data mapping depends entirely on its feature vector. This observation is stated formally in Lemma 2. Lemma 2 is a direct corollary of Lemma 1.

Lemma 1. Let dm be a data mapping such that $FV(dm) = (e_1, \ldots, e_{pf})$. Let q be a query such that $FV(q) = (q_1, \ldots, q_{pf})$, then

$$dm \in DM^+(q) \Leftrightarrow FV(dm) \geq FV(q) \Leftrightarrow \forall i: (e_i \geq q_i)$$

Proof:

Let dm and q be a data mapping and a query q such that $FV(dm) = (e_1, \ldots, e_{pf})$ and $FV(q) = (q_1, \ldots, q_{pf})$.

- Assume $dm \in DM^+(q)$. Let $i \in pf$:

- If $q_i = 0$, since $e_i = 0$ or 1, $e_i \geq q_i$.

- If qi = 1, by definition of *query feature vector*, predicate σ_i appears in query q.

  By definition of $DM^+(q)$, dm does not negate σ_i.

  By definition of example feature vector, $e_i = \sigma_i(dm) = 1$. As a consequence, $e_i \geq q_i$.

  This concludes proof that $dm \in DM^+(q) \Rightarrow \forall i: (e_i \geq q_i)$ and $FV(dm) \geq FV(q)$.

- Assume that $\forall i: (e_i \geq q_i)$.

  Let $j \in pf$, such that predicate σ_j appears in query q.

By definition of query feature vector $q_j = 1$.

In particular, $e_j \geq q_j$. By definition of example feature vector, this means that dm does not negate predicate σ_j.

This concludes proof that dm does not negate any predicates in q, which means that $dm \in DM^+(q)$.

Lemma 2. Let dm and dm' be two data mappings such that FV(dm)=FV(dm'). For any query q

$$dm \in DM^+(q) \Leftrightarrow dm' \in DM^+(q)$$

Proof:

Let dm and dm' be two data mappings such that $FV(dm) = FV(dm') = (e_1, e_2, \ldots, e_{pf})$.

By application of Lemma 1: $dm \in DM^+(q) \Leftrightarrow \forall i: (e_i \geq q_i)$.

By application of Lemma 1: $dm' \in DM^+(q) \Leftrightarrow \forall i: (e_i \geq q_i)$.

2.3. Feature Vector Label

Each feature vector can always be assigned exactly one of three labels: *pos* (positive), *neg* (negative), or *mis* (missing).

Definition: *Correctly Labeled Pair*

A *labeled pair* is a pair (fv, lbl), where fv is a feature vector and lbl is one of the three labels {*neg*, *pos*, *mis*}. Let q be a query, (fv, lbl) is a *correctly labeled pair* with respect to q if and only if all of the following conditions are met:

- If for all data mapping dm in DM, $FV(dm) \neq fv$, then lbl = *mis*

- If there exists a data mapping dm such that FV(dm) = fv and $dm \in DM^+(q)$ then lbl = *pos*

- If there exists a data mapping dm such that FV(dm) = fv and dm$\notin DM^{+}$ (q) then lbl = *neg*

Lemma 1 and Lemma 2 guarantee that the three cases in the above definition are mutually exclusive, and that their disjunction is always true.

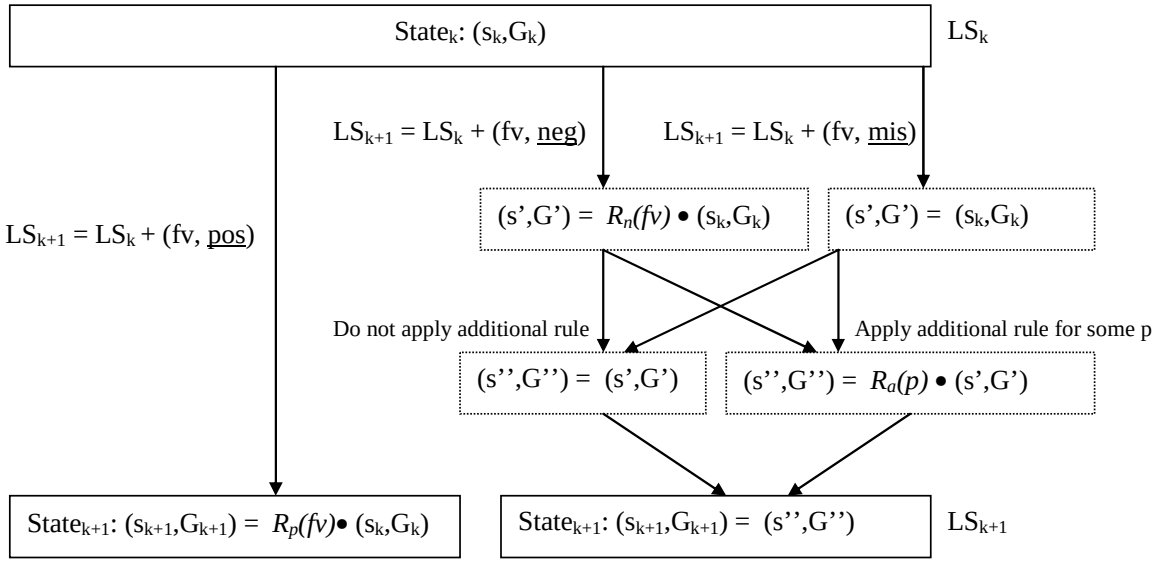


Figure 28 – Sphinx learning algorithm State Transitions

2.4. Learning Algorithm Sequences

Interaction between the user and the system, described in Section 5, results in the user constructing a label sequence, one labeled pair at a time.

Definition: *Label Sequence*

A label sequence is a sequence of labeled pairs.

Two quantities characterize the state of the Sphinx learning algorithm at any moment: the Version Spaces state, and a Label Sequence. Figure 28 illustrates the state transition, when a labeled pair (fv, label) is added to the label sequence. The initial state is characterized by vector (s_k, G_k), and label sequence LS$_k$. The algorithm branches on the label value, and applies some rules, which reduce the space of remaining hypothesis by operating on the Version Spaces state. As shown there are three different rules and

therefore three different kinds of operators exist which operate on the Version Spaces state: the *positive rule* with operator $R_p(fv)$ which is applied in the positive branch, the *negative rule* with $R_n(fv)$ which is applied in the negative branch, and the *additional rule* with $R_a(p)$ which can be applied in both the negative and the missing branch.

Definition: *Rule Sequence*

A rule sequence is a sequence of rule operators, which are of three kinds: $R_p(fv)$, $R_n(fv)$, $R_a(p)$.

Definition: *Rule Sequence Triggered by a Label Sequence.*

A rule sequence RS is triggered by a label sequence LS, if the rule sequence is the set of actions taken by the Version Spaces algorithm in response to the label sequence LS.

Definition: *Version Set*

Let RS be a rule sequence. The version set VS(RS) is defined inductively as:

- $VS(\varnothing) = QS(s_0, G_0)$: the empty rule sequence gives a version set equal to the whole search space.

- $VS(RS) = QS(s_k, G_k) \Rightarrow VS(RS + R_x(y)) = QS(R_x(y)(s_k, G_k))$: adding a new rule to the rule sequence is equivalent to applying the rule operator to reduce the space of remaining hypothesis.

2.5. Correctness for Positive Rule Operator

Lemma 3 states that the *positive rule* operator $R_p(fv)$ removes q from the space of remaining hypothesis (QS) if and only if no data mapping with feature vector fv can be a positive example for q. Thus operator $R_p(fv)$ only removes from the space of remaining hypothesis those queries which are incompatible with the correctly labeled pair (fv,pos).

Lemma 3. Let (fv,pos) be a labeled pair, q a query in QS(s, G) and QS' = $R_p(fv)$(QS).

$$q \notin QS' \Leftrightarrow \forall dm : FV(dm) = fv \Rightarrow dm \notin DM^+(q)$$

Proof:

Assume that fv = (e1, e2, …, epf), and q = (q1, q2, …, qpf), and with

$QS = (s, G)$, $QS' = (s', G')$, such that $G = \{(g_{1,1}, \ldots, g_{1,pf}),$

$(g_{2,1}, \ldots, g_{2,pf}),$

$\ldots$

$(g_{k,1}, \ldots, g_{k,pf})\}$

- $q \notin QS' \Rightarrow (\forall dm : FV(dm) = fv \Rightarrow dm \notin DM^{+}(q))$

Assume dm such that FV(dm) = (e1, e2, …, epf).

$$q \in QS, q \notin QS' \Leftrightarrow (\forall i: q_i \le s_i) \wedge (\exists i: g_i \le q) \wedge ((\exists i: q_i > s_i') \vee (\forall i: \neg(g_i' \le q)))$$

and since $\forall i: g_i = g_i'$:

$$\Leftrightarrow (\forall i: q_i \le s_i) \wedge (\exists i: q_i > s_i') \wedge (\exists i: g_i \le q)$$

$$\Leftrightarrow (\exists i: s_i \ge q_i > s_i') \wedge (q \in QS)$$

and since $s_i > s_i'$, we must have $s_i = 1$ and $s_i' = 0$, which by definition of s' and R_p implies $e_i=0$:

$$\Rightarrow (\exists i: q_i > e_i)$$

$$\Rightarrow dm \notin DM^{+}(q)$$

- $(\forall dm : FV(dm) = fv \Rightarrow dm \notin DM^{+}(q)) \Rightarrow q \notin QS'$

Assume dm such that FV(dm) = (e1, e2, …, epf).

$$dm \notin DM^{+}(q) \Rightarrow (\exists i: q_i > e_i)$$

$$\Leftrightarrow (\exists i: q_i = 1 \wedge e_i = 0)$$

and since $q \in QS$, we have $(\forall i: q_i \le s_i) \wedge (\exists i: g_i \le q)$

$$\Rightarrow (\exists i: q_i = 1 \wedge e_i = 0 \wedge q_i \le s_i)$$

$$\Rightarrow (\exists i: q_i = 1 \wedge e_i = 0 \wedge s_i = 0)$$

and since by definition of s' and R_p, s_i' must be 0 in that case :

$$\Rightarrow (\exists i: q_i = 1 \wedge s_i = 1 \wedge s_i' = 0)$$

$$\Rightarrow (\exists i: q_i > s_i')$$

$$\Rightarrow q \notin QS'$$

■

2.6. Correctness for Negative Rule Operator

Lemma 4 states that the operator $R_n(FV(dm))$ removes q from the space of remaining hypothesis (QS) if and only if dm is not a negative example for View(q).

Lemma 4. Let (fv,pos) be a labeled pair, q a query in QS(s, G) and QS' = $R_n(fv)$(QS):

$$q \notin QS' \Leftrightarrow (\forall dm : FV(dm) = fv \Rightarrow dm \in DM^+ (q))$$

Proof:

Assume that fv = (e_1, e_2, …, e_{pf}) and q = (q_1, q_2, …, q_{pf}) and with

$QS = (s, G)$, $QS' = (s'', G'')$ such that s'' = s

$$G = \{(g_{1,1}, \ldots, g_{1,pf}),$$
$$(g_{2,1}, \ldots, g_{2,pf}),$$
$$\ldots$$
$$(g_{k,1}, \ldots, g_{k,pf})\}$$

$$G'' =$$

$$\{ (g_{i,1}', \ldots, g_{i,pf}') \mid [(g_{i,1}', \ldots, g_{i,pf}') \preceq s'] \wedge$$

$$[\ [(\exists p: (g_{p,1}, \ldots, g_{p,pf}) \in G) \wedge (\forall j: e_j=0 \Rightarrow g_{p,j} = 0)) \wedge (\exists f: (\forall j \neq f: g_{i,j}' = g_{p,j}) \wedge e_f=0 \wedge g_{i,f}' = 1)]$$

$$\vee [(\exists p: (g_{p,1}, \ldots, g_{p,pf}) \in G) \wedge (\exists f: e_f=0 \wedge g_{p,f} = 1) \wedge (\forall j: g_{p,j} = g_{i,j}'))]\ \ ]\ \}$$

- $q \notin QS' \Rightarrow (\ \forall dm : FV(dm) = fv \Rightarrow dm \in DM^{+}(q)\ )$

Assume dm such that $FV(dm) = (e_1, e_2, \ldots, e_{pf})$, since s'' = s we have:

$$q \notin QS' \Rightarrow (\exists j\ \forall i: q_i \geq g_{j,i}) \wedge (\forall j\ \exists i: q_i < g_{j,i}').$$

Let z be such that $\forall i: q_i \geq g_{z,i}$.

Assume A: $(\exists i: e_i = 0 \wedge g_{z,i} = 1)$:

in that case $g_z \in G''$ (by fulfilling the second part of the disjunction),

and because $g_z \leq q \leq s'' = s$, we find that $q \in QS'$ which is impossible.

We therefore deduce $\neg A: (\forall i: (e_i = 0) \Rightarrow (g_{z,i} = 0))$

Let f be any f such that $e_f = 0$, we define the vector $ng = (ng_1, ng_2, \ldots, ng_{pf})$ with $ng_f = 1$, and $(\forall i \neq f: ng_i = g_{z,i})$

Assume B: $s_f = 0$, then because $q \leq s$, we have $q_f = 0$

Assume $\neg B$: $s_f = 1$, then by definition $ng \in G'$

and since $(\forall i \neq f: ng_i = g_{z,i})$, we have: $(\forall i \neq f: ng_i \leq q_i)$

Assume C: $q_f = 1$,

then $ng_f \leq q_f$,

and since $(\forall i: ng_i \leq q_i)$, therefore $ng \leq q \leq s'' = s$

this implies $q \in QS'$ which is impossible.

We can therefore deduce $\neg C: q_f = 0$.

QED (we have just proved $\forall f: e_f = 0 \Rightarrow q_f = 0$)

- $(\forall dm : FV(dm) = fv \Rightarrow dm \in DM^{+}(q)) \Rightarrow q \notin QS'$

Assume dm such that $FV(dm) = (e_1, e_2, \ldots, e_{pf})$.

Let $g'' \in G''$: $g'' = (g_{z,1}', g_{z,2}', \ldots, g_{z,pf}')$ with all the properties listed above for a member of G'', in particular $(\exists p: g_p=(g_{p,1}, \ldots, g_{p,pf}) \in G)$ such that:

$$[(\forall j: e_j=0 \Rightarrow g_{p,j} = 0)) \wedge (\exists f: (\forall j \neq f: g_{i,j}' = g_{p,j}) \wedge e_f=0 \wedge g_{i,f}' = 1)]$$

$$\vee [(\exists f: e_f=0 \wedge g_{p,f} = 1) \wedge (\forall j: g_{p,j} = g_{i,j}')]$$

Take such a p:

Assume A: $(q \geq g_p)$.

Since $dm \in PDM(q)$, therefore we have $(\forall i: e_i \geq q_i \geq g_{p,i})$.

The property $(\exists f: e_f = 0 \wedge g_{p,f} = 1)$ is now impossible, therefore by definition of G'':

$$(\exists f: (e_f = 0) \wedge (g_{z,f}' = 1) \wedge (\forall i \neq f: g_{z,i}' = g_{p,i}))$$

Take such an f: because $(e_f = 0)$ and $(\forall i: e_i \geq q_i)$; the only possibility is in that case that $q_f = e_f = 0$

$(q_f = 0)$ and $(g_{z,f}' = 1)$ imply $\neg(g'' \leq q)$

Assume $\neg$A: $(\exists i: q_i < g_{p,i})$

Assume B: $(\forall j: g_{z,j}' = g_{p,i})$

then $(\exists i: q_i < g_{p,i} = g_{z,j}')$ which implies $\neg(g'' \leq q)$

Assume $\neg$B: $\neg(\forall j: g_{z,j}' = g_{p,i})$

Then by definition of G'', the other part of the disjunction must be true:

$$[(\forall j: e_j=0 \Rightarrow g_{p,j} = 0)) \wedge (\exists f: (\forall j \neq f: g_{z,j}' = g_{p,j}) \wedge e_f=0 \wedge g_{z,f}' = 1)]$$

in particular:

$$(\exists f: g_{z,f}' = 1 \wedge (\forall j \neq f: g_{z,j}' = g_{p,j}))$$

therefore:

$$(\forall j: g_{z,j}' \geq g_{p,j}).$$

Recall that:

$$(\exists i: q_i < g_{p,i}) \Rightarrow (\exists i: q_i < g_{p,i} \leq g_{z,i}') \Rightarrow \neg(g'' \leq q)$$

QED - we have just proved that:

$$(\forall g'' \in G'': \neg(g'' \leq q)))$$

■

2.7. Correctness for Additional Rule Operator

Lemma 5(k) proves that the application of the *additional rule operator* at step k in the rule sequence, only removes from the Version Spaces state redundant target query definition. By making explicit the conditions under which two target queries may define the same target view, Lemma 5(k) guarantees that each view, which could still be the correct *target view*, preserves exactly one representative query in the Query Set. Thus, We state Lemma 5(k), and prove that Lemma 5(k) holds for all values of k in the rule sequence.

Definition: *Compatible*

A query q is *compatible* with a label sequence LS if and only if the following properties are true:

- (fv, pos) $\in$ LS $\Rightarrow$ (fv, pos) is a correctly labeled pair with respect to q
- (fv, neg) $\in$ LS $\Rightarrow$ (fv, neg) is a correctly labeled pair with respect to q

Lemma 5(k). If $LS_k = ((dm_1, l_1), \ldots, (dm_k, l_k))$ is a label sequence, and:

- RS_k is a rule sequence triggered by LS_k such that $(R_a(p_1), R_a(p_2), \ldots, R_a(p_a))$ is the exact subsequence of applications of the *additional rule* in RS_k,
- $VS(RS_k) = QS(s_k, G_k)$,

where s_k and G_k are the most specific and most general boundaries of $VS(RS_k)$

- q is compatible with LS_k,
- q' a query such that:

$$(\forall j \leq a: q_{pj}' = 1) \wedge (\forall i: (\forall j \leq a: i \neq p_j) \Rightarrow (q_i' = q_i))$$

Then:

$$q' \in VS(RS_k)$$

Proof:

Lemma 5 is parameterized by k, the length of the label sequence. The proof is an induction on k.

- k = 0

$LS_0 = \varnothing$, $RS_0 = \varnothing$, a=0. We simply verify that q = q' and that both are in $VS(\varnothing) = QS(s_0, G_0)$ which is the whole search space.

- Assume Lemma 5(k) is true: prove Lemma 5(k+1)

Case 1:

The sub-sequence of applications of *additional rule operators* is the same for RS_{k+1} and RS_k. In other terms there is no application of the *additional rule operator* between step k and step k+1.

Assume q and q' are queries such that q is compatible with LS_{k+1} and q' is such that:

$$(\forall j \leq a: q_{pj}' = 1) \wedge (\forall i: (\forall j \leq a: i \neq p_j) \Rightarrow (q_i' = q_i))$$

We can apply the induction hypothesis, Lemma 5(k) on LS_k, RS_k, q and q': therefore $q' \in VS(RS_k)$.

Assume that $fv = (e_1, e_2, \ldots, e_{pf})$. There are three further cases on the value of the label l_{k+1}:

- If l_{k+1} = pos, then there exists $dm_{k+1} \in DM^{+}(q)$, such that $FV(dm_{k+1}) = fv_{k+1}$ because q is compatible with LS_{k+1}.

  $dm_{k+1} \in DM^{+}(q) \Rightarrow (\forall i: e_i \geq q_i)$

  There are two further cases:

  - Let i be such that $(\forall j \leq a: i \neq p_j)$, then $q_i' = q_i$ and $e_i \geq q_i'$
  - Let i be such that $(\exists j \leq a: i = p_j)$, then

    let j be such that $j \leq a$ and $i = p_j$

    Assume A: $(e_i = 0)$, then $fv_{k+1} \in 0_i$,

    then Precondition for $R_a(i=p_j)$ in the rule sequence, dictates that the label l_{k+1} be negative or missing. This is impossible.

    We can therefore deduce $\neg$A: $(e_i = 1)$, and $e_i \geq q_i'$ is assured.

  Thus we proved with both cases that $(\forall i: e_i \geq q_i')$,

  and therefore that $dm_{k+1} \in DM^{+}(q')$.

  Using Lemma 3, we can deduce that $q' \in R_p(fv_{k+1})(s_k, G_k)$,

  therefore $q' \in VS(RS_{k+1})$.

- If l_{k+1} = neg, then there exists $dm_{k+1} \notin DM^{+}(q)$ such that $FV(dm_{k+1}) = fv_{k+1}$, because q is compatible with LS_{k+1}

  $dm_{k+1} \notin DM^{+}(q) \Rightarrow (\exists i: e_i < q_i)$.

  Let i be such that $e_i < q_i$. There are two further cases:

  - $(\forall j \leq a: i \neq p_j)$, then $q_i = q_i'$ and $e_i < q_i'$. Therefore $dm_{k+1} \notin DM^{+}(q')$
  - $(\exists j \leq a: i = p_j)$, then let j be such that $j \leq a$ and $i = p_j$.

    $e_i < q_i \Rightarrow e_i = 0$, and since $q_i' = 1$, $e_i < q_i$ is assured. Therefore $dm_{k+1} \notin DM^{+}(q')$.

  With both cases we established $dm_{k+1} \notin DM^{+}(q')$.

  Using Lemma 4, we can deduce that $q' \in R_n(fv_{k+1})(s_k, G_k)$,

  and since we are in the case where there is no application of the *additional rule* between step k and step k+1: $q' \in VS(RS_{k+1})$

- If l_{k+1} = mis, then since there is no application of the *additional rule*, $RS_k = RS_{k+1}$
  and q' $\in$ $VS(RS_{k+1})$

Case 2:

The subsequence of applications of the *additional rule operator* is incremented from RS_k to RS_{k+1} by the application of $R_a(p_{a+1})$. In other terms $R_a(p_{a+1})$ is applied between step k and step k+1.

Assume q and q' are queries such that q is compatible with LS_{k+1} and q' is such that:

$$(\forall j \leq a+1: q_{pj}' = 1) \wedge (\forall i: (\forall j \leq a+1: i \neq p_j) \Rightarrow (q_i' = q_i)).$$

Let q'' = (q_1'', q_2'', …, q_{pf}'') be such that:

$$(\forall j \leq a: q_{pj}'' = 1) \wedge (\forall i: ((\forall j \leq a: i \neq p_j) \Rightarrow q_i'' = q_i)).$$

We can apply the inductive hypothesis, Lemma 5(k) to q'':

$$q'' \in VS(RS_k).$$

Assume that $fv_{k+1} = (e_1, e_2, \ldots, e_{pf})$. There are two further cases:

- If l_{k+1} = neg, then there exists $dm_{k+1} \notin DM^+(q)$ such that $FV(dm_{k+1}) = fv_{k+1}$, because q is compatible with LS_{k+1}
  $dm_{k+1} \notin DM^+(q) \Rightarrow (\exists i: e_i < q_i)$
  There are two further cases:
  - Let i be such that ($\forall j \leq a: i \neq p_j$), then $q_i'' = q_i$ and therefore $e_i < q_i''$
  - Let i be such that ($\exists j \leq a: i = p_j$), then
    let $j \leq a$ such that $i = p_j$: in that case $q_i'' = 1$
    and since $e_i < q_i$. The only possibility is $e_i = 0 < q_i'' = 1$.

  Thus we proved with both cases that:

$$(\exists i: e_i < q_i'')$$

$$\Rightarrow dm_{k+1} \notin DM^{+}(q'').$$

Using Lemma 4:

$$q'' \in R_n(fv_{k+1})(s_k, G_k).$$

Note that:

$$(\forall i \neq p_{a+1}: q_i'' = q_i') \wedge (q_{pa+1}' = 1).$$

There are two cases:

- Case A: $q_{pa+1} = 1$: in this case since $q_{pa+1}''=q_{pa+1}$, we have q'' = q', and since $q_{pa+1}'' = 1$, by definition of the action for $R_a(p_{a+1})$:

  $$q'' \in R_a(p_{a+1})(R_n(fv_{k+1})(s_k, G_k))$$

  Therefore $q' \in VS(RS_{k+1})$.

- Case B: $q_{pa+1} = 0$: $q_{pa+1}'' = 0$, $q_{pa+1}' = 1$

  We will prove that q' is compatible with LS_{k+1}, then we will deduce that $q' \in VS(RS_{k+1})$.

  - Let $i \leq k+1$ be such that $(fv_i, l_i = pos) \in LS_k$.

    There exists $dm_i \in DM^{+}(q'')$ s.t. $FV(dm_i) = fv_i = (f_1, \ldots, f_{pf})$

    $dm_i \in VM(q'') \Rightarrow (\forall i: f_i \geq q_i'')$.

    If $f_{pa+1} = 1$ then $(\forall i: f_i \geq q_i') \Rightarrow dm_i \in DM^{+}(q')$

    If $f_{pa+1} = 0$ then by the precondition for $R_a(p_{a+1})$, l_i is neg or mis, which is impossible.

    Therefore $dm_i \in DM^{+}(q')$.

  - Let $i \leq k+1$ be such that $(fv_i, l_i = neg) \in LS_k$

    There exists $dm_i \notin DM^{+}(q'')$ s.t. $FV(dm_i) = fv_i = (f_1, \ldots, f_{pf})$

    In this case $dm_i \notin DM^{+}(q'')$, which implies that $\exists i: f_i < q_i''$.

    If $i = p_{a+1}$ then since $q_{pa+1}'' = 0$, there is a contradiction.

  If $i \neq p_{a+1}$. In that case: $q_i'' = q_i'$, which implies that $f_i < q_i''=q_i'$.

  Therefore $\exists i: f_i < q_i'$ and $dm_i \notin DM^{+}(q')$.

We have just established that q' is compatible with LS_{k+1}.

By induction on the label sequence LS_k, we now prove that q' $\in$ $VS(RS_{k+1})$:

- q' $\in$ VS(∅). q' is in the initial search space.
- assume q' $\in$ $VS(RS_i) = QS(s_i, G_i)$, RS_i triggered by LS_i, $LS_{i+1} = LS_i + (fv_i, l_i)$
  - If l_i = pos, since q' is compatible with LS',

    there exists $dm_i \in DM^+(q')$ s.t. $FV(dm_i) = fv_i$

    and by Lemma 3, q' $\in$ $VS(RS_i + R_p(fv_i))$
  - If l_i = neg, since q' compatible with LS',

    there exists $dm_i \notin DM^+(q')$ s.t. $FV(dm_i) = fv_i$

    by Lemma 4, q' $\in$ $R_n(fv_i,)(s_i, G_i)$.

    $\forall j \le a+1$: $q_{pj}' = 1$ and by definition of the $R_a(p_j)$: ($\forall j$: q' $\in$ $R_a(p_j)(R_n(fv_i)(s_i, G_i))$.

    Since by definition of the algorithm, when l_i = neg, RS_{i+1} is equal to either $RS_i + R_n(fv_i)$ or $RS_i + R_n(fv_i) + R_a(p_j)$ for some j:

    q' $\in$ $VS(RS_{i+1})$.
  - If l_i=mis, then

    $\forall j \le a+1$, $q_{pj}' = 1$, and by definition of the $R_a(p_j)$ operator, we have $\forall j$: q' $\in$ $R_a(p_j)(s_i, G_i)$.

    Since by definition of algorithm, RS_{i+1} is equal to either RS_i or $RS_i + R_a(p_j)$ for some j:

    q' $\in$ $VS(RS_{i+1})$.

- If l_{k+1} = mis, there are two cases:
  - If q_{pa+1} = 1, then q' = q''.

  q' = q'' $\in$ $VS(RS_k)$,

  and since q_{pa+1}'=1, by definition of $R_a(p_{a+1})$:

  q' $\in$ $R_a(p_{a+1})(s_k, G_k) = VS(RS_{k+1})$.

- If $q_{pa+1}' = 0$. Same scenario and same proof as in Case B above. We first prove that q' is compatible with LS_{k+1}, then by a mini-induction that $q' \in VS(RS_{k+1})$. QED

■

2.8. Correctness and Termination of the Sphinx Learning Algorithm.

Theorem 1 and Theorem 2 together guarantee eventual convergence of the Sphinx learning algorithm towards a single view definition consistent with all labeled example examples, provided that a target query was included in the initial search space. Correctness of the result produced by the learning algorithm is summed up in Theorem 1. Eventual convergence towards a result is summed up in Theorem 2.

Theorem 1 states that, if the target view is in the initial search space, the learning algorithm always preserves a representative query defining that view in the space of remaining hypothesis. In particular, if Sphinx has converged to a single query, then that query correctly defines the target view. Theorem 1 is a direct corollary of Lemma 5(k) for all values of k. Note that it is possible for the algorithm not to converge to a single query and to yield the empty set when there are no remaining hypotheses consistent with the labeled examples. This will only happen when the target query is not in the initialized search space.

Theorem 1 (Correctness). Let vt = TQ(DB) be the view defined by a query TQ over database DB,

such that TQ is in the original hypothesis space and $FV(TQ) = (q_1, q_2 \ldots, q_{pf})$,

let LS be a label sequence such that q_t is compatible with LS,

and let RS be the rule sequence triggered by LS.

Then $\exists q \in VS(RS)$ such that the view defined by q, $DM^+(q)$, is equal to v_t.

Proof:

Assume q is a representative such that $DM^{+}(q) = v_t$, and such that q is eliminated at step k in the rule sequence. Since $v_t = DM^{+}(q)$, q is consistent with all positive and negative examples, and Lemma 3 and Lemma 4 prove that q cannot be eliminated from the search space by an application of the *positive rule* or the *negative rule operator*. Therefore q can only be eliminated by application of the *additional rule operator* at some step k.

Let $(R_a(p_1), R_a(p_2), \ldots, R_a(p_a))$ be the exact subsequence of applications of the *additional rule* operator in RS_k. Lemma 5(k) identifies q' such that $(\forall j \leq a: q_{pj}' = 1) \wedge (\forall i: (\forall j \leq a: i \neq p_j) \Rightarrow (q_i' = q_i))$, and such that q' is in VS_k.

Assume that $dm \in v_t$, such that $FV(dm) = (e_1, e_2, \ldots e_{pf})$. By application of Lemma 1, $\forall i: e_i \geq q_i$.

Let i, such that $\forall j \leq a: i \neq p_j$. By definition of q': $q_i' = q_i$ and $e_i \geq q_i'$.

Let i such that $\exists j \leq a: i = p_j$. By definition of q': $q_i' = 1$.

Since $dm \in v_t$, dm cannot be a negative example. By definition of additional rule precondition $R_a(i)$: $FV(dm) \notin 0_i$. Therefore $e_i = 1$ and $ei \geq qi' = 1$.

This concludes proof that $v_t \subseteq DM^{+}(q')$.

Assume that $dm \in DM^{+}(q')$. By definition of q', $\forall i: (q_i' \geq q_i)$, and by application of Lemma 1, this implies that $dm \in DM^{+}(q) = v_t$.

This concludes proof that q' is such that $v_t = DM^{+}(q')$.

■

Ignoring missing example instances, and applying the original version spaces algorithm trained solely with positive and negative examples can lead to a deadlock situation. Such a deadlock occurs when two target queries TQ_1 and TQ_2, define the same target view TV over the source databases ($DB(TQ_1) = DB(TQ_2) = TV$). When this is the case the version spaces will not converge since both TQ_1 and TQ_2 will be consistent with

the training set, and the examples necessary to discriminate between then are all be labeled missing. Theorem 2 states that Sphinx, will eventually converge to a single query and avoid deadlock. To discuss the issue of convergence, and prove Theorem 2, it is necessary to introduce the notion of *partial convergence on a potential feature*.

Definition: *Partial Convergence on a Potential Feature*

Assume the Version Spaces state is (**s**, **G**), with s = (s_1, s_2, …, s_{pf}), G = {($g_{1,1}$, $g_{1,2}$, …, $g_{1,pf}$)…($g_{k,1}$, $g_{k,2}$, …, $g_{k,pf}$)}. The Sphinx learning algorithm has partially converged for potential feature F_i, if and only if: $s_i = g_{1,i} = g_{2,i} =… = g_{k,i}$.

If Sphinx has partially converged for all potential features, then it has converged (in the usual sense) and the target query is known.

Theorem 2 states that if each of the 2^{pf} distinct feature vectors is assigned a label, the algorithm is guaranteed to have converged to a solution. Theorem 2 proves that successive applications of the *Additional Rule operator* remove all redundant query definitions in the Version Space state, leaving only one representative of the target view that the algorithm converges on.

Theorem 2 (Termination). Let v_t = TQ(DB) be the view defined by a query TQ over database DB, such that TQ is in the original hypothesis space and FV(TQ) = (q_1, q_2…, q_{pf}).

Let LS be a label sequence such that LS contains 2^{pf} distinct, correctly labeled pairs w.r.t. v_t, one each for every possible feature vector instance.

Let RS be the rule sequence triggered by LS,

Then VS(RS) = QS(s, G) has converged to a single query.

Proof:

In order to prove that VS(RS) has converged to a single query, we will prove that partial convergence has been reached for each of the potential features.

Since TQ is in the original search space, by application of Theorem 1, at any step k, VS_k always contains at least one element and the algorithm continues until every element in the label sequence LS has triggered the application of a rule.

Let f ≤ pf, be a potential feature:

• Assume A: [$\forall$i: (((dm$_i$ = (e$_1$, e$_2$, …, e$_{pf}$), l$_i$) $\in$ LS) $\wedge$ (e$_f$ = 0)) $\Rightarrow$ l$_i$ = neg or mis]

Since LS contains all possible feature vectors, the precondition for $R_a(f)$ is fulfilled.

Since every element of LS is guarantee to trigger the application of a rule, $R_a(f)$ is guaranteed to be in the rule sequence RS.

By definition of $R_a(f)$: ($\forall$i: s$_f$ = g$_{i,f}$ = 1).

The algorithm has partially converged on feature f.

• Assume $\neg$A: [$\exists$i: (((dm$_i$ = (e$_1$, e$_2$, …, e$_{pf}$), l$_i$) $\in$ LS) $\wedge$ (e$_f$ = 0)) $\wedge$ l$_i$ = pos]

Since every element of LS is guarantee to trigger the application of a rule, there exists i such that $R_p(FV(dm_i))$ is in the rule sequence RS.

Since e$_f$ = 0, by definition of $R_p(FV(dm_i))$: ($\forall$i: s$_f$ = g$_{i,f}$ = 0).

The algorithm has partially converged on feature f.

This completes proof that the algorithm has converged on all potential features.

■

3. Active Learning and Sample Selection

Active Learning refers to a process where the system selects examples for the user to label, in order to reduce the cost of labeling unnecessary examples. In this section we look at the problem of selecting the examples that must be submitted to the user.

The number of examples necessary to converge to an answer is an important measure of success for Sphinx and active-learning systems in general. The goal is to

converge with a minimal number of examples. The fewer examples, the better the user experience. It should be noted that in an adversarial worst-case scenario, the system could be forced to test 2^{pf}-(pf choose 2) negative or missing examples, before the *additionalrule* could be used to converge. Thus the potential of system performance and the opportunity to introduce heuristics are vast.

3.1. Active Learning as a Search Problem

To explore its search space, Sphinx proceeds incrementally by trying to label a series of intermediate sub-goals. The active learning portion of Sphinx chooses the examples it presents to the user with the purpose of achieving its current sub-goal. Consider the feature vector illustrated in Figure 29. All its feature bits are set to 1 except for F_1. A data-mapping instance with this example feature vector would negate only predicate σ_1.

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}
0	1	1	1	1	1	1	1	1	1	1	1	1

Figure 29 – Example Feature Vector

Consider the data mapping shown in Figure 30. Its example feature vector is equal to the one shown in Figure 29. This data mapping is similar in all points to the data mapping shown in Figure 19, except for a minor difference in the value of Product_ID. This minor difference results in the negation of predicate σ_1.

If the user assigns this data mapping a positive label then Sphinx will know σ_1 cannot be a filter predicate in the target query. Conversely if the user assigns it a negative label then the target query must contain filter predicate σ_1. Thus, regardless of the actual label value, Sphinx can ascertain whether or not σ_1 belongs to the target query. If the same mechanism is applicable to the 12 other potential features, Sphinx could in this particular instance converge in 13 steps.

3.2. The Role of Functional Dependencies

Let us consider carefully the proposed strategy, which by setting sub-goals for partial convergence, would proceed incrementally to full convergence. This strategy seeks to find data mapping instances for each of the 13 sub-goals shown in Figure 31, and then to submit them to the user. To reach these sub-goals, a data mapping instance with example feature vector equal to the sub-goal must be submitted to the user (a query on the source can easily be written to retrieve data mapping instances corresponding to a given feature vector representation). This strategy is particularly attractive, because if the necessary 13 data mappings can be found, regardless of how the user labels them, the algorithm fully converges. However in a practical scenario, using real data sources, none of the data mappings necessary for any of the sub-goals in Table 10, are likely to exist. These sub-goals are too specific: they require data mappings with very restrictive integrity constraints. The reader can observe how unlikely the system is to find a row in Product Review, which would produce the mapping shown in Figure 30: if the Product_ID differs from '301-001' to '301-002', it is likely that a lot more description and key fields will differ as well.

Generally, because of functional or accidental dependencies in the data and because many predicates will not be independent (e.g. σ_1 and σ_3) most sub-goals are unreachable and looking for the corresponding data mappings will result in the production of missing labels. Since missing labels do not efficienty lead to convergence, sub-goals must be chosen carefully. In particular at the beginning of the search, catalog information allows us to identify "tautological" predicates that are never negated, and exclude them from version spaces. Such predicates may originate from fields that are left unfilled or always default to the same value. For example the tautological predicate 'phone = 000-0000' may appear if phone numbers in the database were never collected.

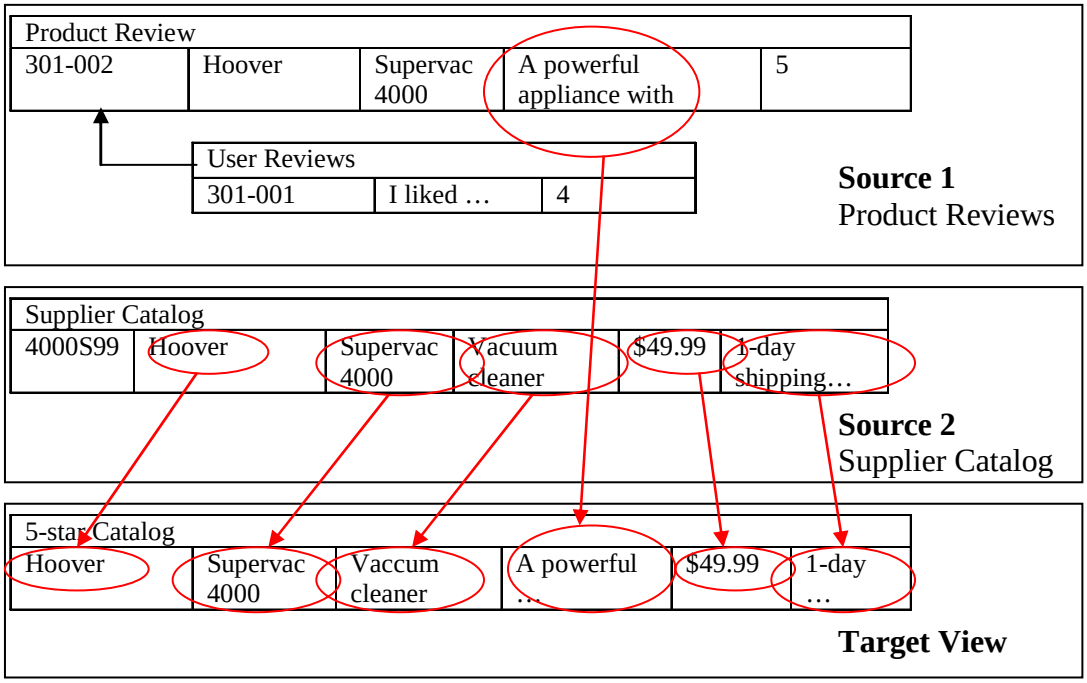


Figure 30 – Data Mapping Instance with the Example Feature Vector shown in Figure 29

Sub-goal 1: (0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
determine if σ_1 must be included in the target query
Sub-goal 2: (1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
determine if σ_2 must be included in the target query
…
…
Sub-goal 13: (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0)
determine if σ_{13} must be included in the target query

Figure 31 – Sequence of sub-goals leading to convergence

3.3. Value of Positive Examples vs. Negative and Missing Examples

Since a naïve strategy of proving or disproving each individual feature separately (as in Figure 31) cannot be seriously considered, it is important to understand that the value of a positive example is much higher than the value of a negative example. Consider a sub-goal that seeks (to prove or disprove) a group of k filter predicates. Finding a positive data mapping example for that sub-goal, will bring the algorithm closer to convergence: by generalizing the most specific feature vector on all k features, the algorithm automatically reaches partial convergence on those k features.

On the other hand a negative label for the same sub-goal does not bring the algorithm much closer to converging. Because all k features are negated at the same time, it is not the case that the negative rule will achieve partial convergence on any feature. The nature of the conjunctive language of hypothesis is such that it admits a least upper bound. Positive example by directly altering that least upper bound, eliminate more candidates.

Missing labels can only be exploited in the context of the *additional rule*. Despite Theorem 2 guaranteeing convergence, the algorithm requires a large number of missing or negative labels for a single application of the *additional rule* (even with optimizations we introduce later). Identifying missing labels require queries against the source databases, which are expensive. Negative examples require submitting an example to the user for labeling. Both operations are costly. The additional learning rule does also knock out large numbers of candidates, guaranteeing partial convergence on all k features at once. However, the additional rule requires an exponential number of labels to be applied, and its efficiency in terms of number of candidates eliminated per example instance is exactly equal to that of the negative rule.

Thus, in strict quantitative terms, positive examples are much more useful to eliminate candidates from the search space. In addition, we note that the number of potential filter predicates that can be invented for any given query (even limited to equality), is at least an order of magnitude larger than what will appear in any given query. Thus, if this instance of Occam's Razor is correct, we can state that a lot more generalizing is necessary than specializing: the target query is much more likely to lie close to the most general boundary, than to the most specialized (least upper bound). As expressed above, generalizing the most specific boundary requires positive examples.

3.4. Scarcity of Positive Examples

We have established that positive examples are both necessary and efficient in terms of convergence. Additionally, we examine the relative abundance of positive versus negative examples, and of negative versus missing examples.

Even large databases are sparse when considered against the space of example feature vectors. The latter is measured at 2^{pf}, and even with pf=30 can reach 10^9, a size only the largest databases can hope to attain. Skew makes it likely that many database rows share identical feature vectors. In particular data mappings produced by a Cartesian product will fit in the same number of slots as a natural join, which is a much smaller set. This means the set of feature vectors actually corresponding to examples that can be retrieved from the sources is sparse in the space of example feature vectors. Most elements in that space will be labeled missing. Only a small fraction will correspond to actual examples taken from the data that can be labeled positive or negative.

Next, we seek to evaluate the ratio of example feature vectors for positive examples to example feature vectors for any existing data mappings: $DM^+(TQ)/DM(TQ)$.

- If the target query TQ contains no filters in the Where clause, then this ratio equals 1. This means that negative examples will be impossible to find. Fortunately in that scenario, none are necessary to converge the learning algorithm.
- If the target query contains exactly one filter, for example a natural join predicate, then this ratio is one half: positive and negative examples are equally likely to be found based on their example feature vectors.
- If the target query contains two or more filters, this ration is less than one half, and negative examples will always be much easier to find than positive examples.

Thus, we conclude that in addition to be valuable because of their efficiency in reducing the size of the search space, positive examples are valuable because of their relative rarity.

3.5. Sample Selection Queries

We must stress that the size of the search space does not allow for a methodical classification of all potential examples with respect to all potential remaining hypothesis:

- The number of remaining concepts at any given time is exponentially large.

- The number of potential examples is linked to the size of Cartesian products over the database and also extremely large.

Thus, we must proceed heuristically to find the best possible sample to present to the user at each given moment.

In general a sample corresponding to a given feature vector can be found in the source data by issuing the simple query derived from the feature vector. Because for a given feature vector the probability of finding a sample existing in the database is fairly low, this is not feasible in practice for individual feature vectors.

Thus we issue more general queries. There are 3 possibilities: queries which are neutral on certain features, positive on others, and negative on the rest: so we need a 3 valued query vector to perform actual sample selection.

3.5.1. Three Valued Vector

Given a three-valued vector $tv = (v_1, v_2, \ldots, v_{pf})$, it is possible to build the query q such that for all i, if $e_i = -1$, predicate $\neg\sigma_i$ appears in query q, if $e_i=1$ then σ_i appears in query q, and if $e_i=0$ then neither σ_i nor $\neg\sigma_i$ appears in q. This query q can be run against the source data and will return the result $DM^+(q)$.

3.5.2. Finding Positive Examples

To find positive examples, we must choose a query q such that $DM^+(q) \subseteq DM^+(TQ)$ where TQ is the target query. Keeping in mind Lemma 1, if we believe that predicate σ_i is likely to appear in TQ, and we are looking to submit a positive example to the user, we must choose q so that σ_i =1, and choose v_i =1. On the other hand if the probability that predicate σ_i appears in TQ is negligible, then the presence of σ_i is not likely to change whether or not an element of $DM^+(q)$ is a positive example. We would then choose $v_i = 0$.

For a randomly selected element of $DM^+(q)$ to be a positive example, q must be chosen such that:

- For predicates, which are certain to appear in the target query, v_i must be 1
- For predicates, which are likely to appear in the target query, v_i should be 0 or 1
- For predicates, which are not likely to appear in the target query, v_i should be 0, or -1
- For predicates, which are extremely unlikely to appear in the target query, v_i should be –1.

Our heuristics will respect those four principles, but adopt various strategies as to the size and composition of the assignment groups.

3.5.3. Finding Negative Examples

Let TQ be the target query such that FV(TQ) = (t_1, …, t_{pf}). Assume that Version Spaces has partially converged on features 1-k: such that t_1, …, t_k are known. If there exists i>k such that $t_i = 1$, then a query q such that FV(q) = (q_1, …, q_{pf}), and $q_i=t_i$ for i≤k, and such that there exists i > k such that $q_i < t_i$ is guaranted and $DM^+(q) \subseteq DM^-(TQ)$. If nonempty the answer set will contain examples labeled negative.

An exhaustive search of $2^{(pf-k)}$ will either result in the application of a rule or in the finding of a positive example.

3.5.4. Finding Negative or Positive Examples

Whether an example with feature vector (e_1, e_2, …, e_{pf}) is labeled positive or negative by the user, the application of either the positive or negative rules will reduce the size of the search space based on the number of features F_i for which $e_i = 0$. In particular, unless there is at least one feature F_i such that $e_i = 0$ there is no progress by application of a rule. However for a query q, with three-valued vector vv, the more 0 bits vector vv contains, the more likely that q will return a nonempty result (as opposed to a missing label).

3.6. Sample Selection Heuristics

In this Section, we look at heuristics to implement sample selection for the Sphinx system. When choosing sub-goals to retrieve and identify examples for induction, we need to consider quantitative factors:

- missing labels are of little value, but are extremely easy to find
- negative labels are also of little value, and the easier to find, the less their value
- positive labels are both valuable and hard to find

3.6.1. Two Phase Heuristic

Any strategy attempting to minimize user interaction, must focus on finding at least one and possibly several examples labeled *positive* by the user. Therefore the basis of our strategy is to separate active learning into two phases. In a first phase Sphinx will exclusively focus on proposing data mappings that it judges are likely to become positive examples. The goal in this first phase is to converge quickly on as many features as possible. A positive example, if it is to bring progress to version spaces, must be found in the pool of hypotheses that have a feature vector more specific than the target query but less specific than the most specific feature vector.

Negative examples that would bring progress to version spaces must also be taken from a pool of hypotheses delimited by the most general boundary on one side, and the target query on the other.

We define a second learning phase where Sphinx will focus on finding data mappings regardless of their likelihood or being positive or negative. In the second phase Sphinx proposes those data mappings that have no particular reason to be positive or negative. In the second phase, the search space is much reduced and as a result finding positive examples, which bring new information is much harder. The expected convergence rate in the second phase will be minimal, but with the extent of the search space reduced by the first phase, the number of examples overall can be contained. As a

first approximation, our heuristics will arbitrarily proceed on to the second phase after one or after two positive examples are found. The question of finding a proper cutoff point between the first and the second phase might merit further study.

3.6.2. Join Feature Bias

A simple analysis shows that the overwhelming majority of the potential features are selection predicates. A selection predicate is created for every attribute in the schema. A join feature is created only when supported by the initial data-mapping example: a relationship exists between two objects in the data mapping.

		Potential Features	Selection Features	Join Features
Healthcare	Query 1	14	14	0
	Query 2	15	15	0
Sports Statistics	Query 3	25	24	1
	Query 4	25	24	1
5 Star Catalog	Query 5	30	28	2
	Query 6	30	28	2
SMD → Base	Query 7	57	57	0
	Query 8	35	33	2

Table 1 – Selection vs. Join Features

An accidental match between a “9.99” as the price $9.99 and a “9.99” as September 99 is possible, but is an unlikely event given any pair of objects chosen at random by the user. Thus almost all join features observed in the data-mapping example are not flukes and do represent existing semantic relationships. Thus two factors combine here to privilege join features: a pure Cartesian product without a join predicate is a very unlikely operation, and most observed join features between objects belonging to different tables are not accidental.

We elaborate a baseline strategy S_b based on this observation. S_b privileges the search for positive examples in its initial phase by initially never choosing sub-goals

negating join predicates. When searching for a sub-goal, strategy S_b looks to negate only a small set of selection predicates (never more than 10). That search is repeated by modifying the set until a positive example is found or the algorithm converges. The small set of negated selection features is chosen with an initial randomization and modified in an incremental search pattern. There is a possibility for backtracking to re-randomize the set, but in the course of running the experiments shown in Table 11, backtracking with S_b occurred only once. Once positive examples have been found, S_b enters its second phase, in which both join and selection predicates will be negated.

It should be observed that in addition to its bias for join features vs. selection features, strategy S_b possesses another built-in bias: it consistently bets that of all the potential features (a large number), only a very small number is likely to actually appear in the *Where* clause of the target query. The latter bias is consistent with our previous observation that only a small portion of all legal predicates are likely to appear in the target query.

3.6.3. Information Gain Bias

We seek a further bias to predict which predicates are more likely to appear in the cardinality component of the target query. We observe that not all features are equally likely. Consider the following feature predicate: "Name = 'Supervac 4000'". It is unlikely to appear in a view defining query since few objects in the source, perhaps only one, will fulfill that predicate, making it useless for any view definition. On the other hand a feature such as "manufacturer/state = 'TX'" is more likely. A significant proportion of the objects may well fall in the 'TX' category and building a view with those might be of use.

We can measure for each selection predicate their information gain. Our basic assumption is that the information gain will serve to estimate the relative likelihood of a predicate with respect to all others. The information gain IG(σ) is a function of the filter factor FF(σ), and is maximal when FF(σ) = 0.5.

$$IG(\sigma) = -(|S_1|/(S_1|+|S_2|))ln(|S_1|/(|S_1|+|S_2|)) - (|S_2|/(|S_1|+|S_2|))ln(|S_2|/(|S_1|+|S_2|))$$

IG(σ) will yield high scores for predicates on enumerated types, and low scores for predicates on infinite types. We note the information gain metric IG(σ), relies on the same catalog statistics used in those query cost models. It has been shown these statistics can be derived even in distributed systems where catalog information is not directly accessible; see (Haas, et al. 1997; Tomasic, et al. 1996). Thus, we are confident we can always rely on such statistics to drive our heuristics.

We introduce a new strategy S_c, a refinement of S_b based on this information gain bias. This strategy does not require likelihood estimations to be accurate. The likelihood function should, above all, cluster predicates into two major categories: the most unlikely predicates (extremely low information gain and infinite domain), and the other predicates (low to high information gain and enumerated domain). This clustering will replace the random process used in S_b.

A comparable bias could be introduced to estimate the relative likelihood of individual join predicates, however as discussed earlier the small number of join predicates precludes the need.

4. Optimizations

In order to clarify the formal presentation, a simple version of the *additional rule* was introduced. In practice, the Sphinx prototype uses a complex *additional rule*, incorporating the following two optimizations: induction of negative examples and application of the rule to partially converged search spaces.

4.1. Induction of Negative Examples

It is immediately apparent that if an example e_1 with feature vector fv_1 is labeled negative, then any example e_2 with feature vector fv_2 such that $fv_1 > fv_2$ cannot be labeled positive. Such a label would not be consistent with the previous negative label. Thus the correct label for fv_2 is either negative or missing. This observation allows us to expand the generality of the Additional Convergence Rule.

4.2. Modified Additional Rule

The additional convergence rule allows the algorithm to partially converge the hypothesis space on p when precondition $R_a(p)$ is met. However because of data dependencies, it is practical to consider a more general form of the additional rule. The heuristic strategy used by the prototype, which is to seek positive examples for a certain cluster of features, requires the application of this modified additional convergence rule. The modified precondition will be met as a side-effect of the heuristic search for positive or negative examples.

Definition: *Set 0{i1, …, ik}*

The set $0\{i_1, \ldots, i_k\}$ is the set of feature vectors containing a zero at all the $i_1, \ldots i_k$ positions:

$$0\{i_1, \ldots, i_k\} = \{ (e_1, e_2, \ldots, e_{pf}) \mid e_{i1} = 0, \ldots, e_{ik} = 0\}$$

Definition: *Precondition $R_a(i1, \ldots, ik)$*

Precondition $R_a(i_1, \ldots, i_k)$ is met, if for any data mapping *dm*, $FV(dm) \in 0\{i_1, \ldots, i_k\}$ implies that dm is a negative example. If there is no data mapping dm, such that $FV(dm) \in 0_p$, then *Precondition $R_a(i_1, \ldots, i_k)$* is true.

$$(\forall dm : FV(dm) \in 0\{i_1, \ldots, i_k\} \Rightarrow dm \text{ is a negative example}) \Rightarrow R_a(i_1, \ldots, i_k)$$

Definition: *Modified Additional Rule Operator (Applied when Precondition Ra(i1, …, ik) is true)*

$R_a(i_1, \ldots, i_k)$: $(s, G) \rightarrow (s', G')$

with $G' = G$

and $s' =$

$$(s_1', \ldots, s_{pf}'), s_{i1}' = 1 \wedge \ldots \wedge s_{ik}' = 1 \wedge (\forall i \neq i_1, .., i_k: s_i' = s_i)$$

5. Adapting Machine Learning Algorithms for our Sample Selection Approach.

The challenges of our approach are threefold: firstly, we require the system to perform sample selection from a vast database of potential examples rather than from a small training set. Secondly, potential example instances can be labeled “missing” and thirdly, the system is required to converge to a correct solution.

The Sphinx algorithm adopts Mitchell’s candidate elimination limited to a very simple concept language using nominal attributes and equality constraints. To adapt the algorithm to the sample selection requirement in relational databases, we defined a new type of label: missing, and defined a watermark rule, enabling us at certain junctures to map missing labels to negative labels to progress convergence towards a target concept with certain properties with respect to missing examples.

Let us examine the possibility of adopting inductive logic programming (ILP) rather than version spaces as a learning system. Inductive logic programming (ILP) is concerned with the induction of logic programs from examples. Relational queries or views are naturally expressed in Datalog, which is essentially Prolog without function symbols. It is possible to consider an inductive logic programming solution for our query discovery problem.

The problem in using current ILP algorithms is how to detect convergence and compute the value of the *(converged?)* function. The *(converged?)* function, when it returns true, terminates active learning with version spaces [Hirsh92]. With systems such as such as Foil [Quinlan90] and Golem [MF90] new rules are created to cover maximal sets of positive examples, while maintaining the exclusion of negative examples. This is done by either creating general rules, which are then specialized (top-down), or by creating specialized rules, which are then generalized (bottom-up).

Here is how ILP would work in a Datalog context: recall that relational predicates represent tables. The available relational predicates are identified as the existing source

tables, additional arbitrary predicates are available (from the user), and equality predicates and unification (substitution of quantified variables with constants) are also available. This forms a concept language equivalent to SQL and which can be restricted in some fashion regarding properties such as self-joins, hidden joins, arbitrary predicates, etc… Further, a classic Datalog program is in DNF. Separate rules represent a natural disjunction, and the body of each rule represents a conjunction of terms. By limiting an ILP algorithm to learning a single rule instead of several, we would restrict the language to purely conjuntive concepts.

We examine the dynamic behavior of the classic ILP approach: the algorithm seeks to maximize cover and keep learning until all positive but no negative examples are covered. This is a stable stopping point for ILP algorithms. A state in which all positive examples are covered and all negative examples excluded does not meet convergence criteria for version spaces. In an active learning approach such dynamic behavior is not acceptable: the start state, which has one positive example given by the user and no negative examples in the training set, would always be the end state. The dynamic behavior of ILP can be altered to conform to the requirements of an active learning algorithm. Consider the dynamic policy suggested by an active learning system such as Sphinx: further examples should be sought whenever there might be an undiscovered example, which taken into account would alter the result of the search. This coincides exactly with the bias-free learning policy of version spaces: as long as the search space is not reduced to a single element, the bias free learner cannot output a result, and the search must progress. One can imagine accommodating active learning with ILP by changing its dynamic behavior in that fashion. Further one may note that the Golem algorithm, which creates maximally specific rules covering given pairs of positive examples has strong affinities with the maximally specific boundary set for version spaces. The Foil algorithm, which creates maximally general rules, and specializes them to exclude negative examples, has strong affinities with the maximally general boundary set for version spaces. In this context adapting the ILP approach by adopting the dynamic

behavior required by an active learner would simulate version spaces, merely substituting Datalog with SQL syntax.

6. Conclusion

We start by decomposing the typical relational query into two distinct components. A structure component regroups relational variable definitions and set projection operators. A cardinality component represents filters and regroups joins and selection relational operators. We showed how the structure component is uniquely determined by a single strong example, and we observe that the cardinality component can be modeled as a standard classifier. We presented two methods of initializing a search space to learn the cardinality component of the query.

We addressed the problem posed by introducing active learning and sample selection in the realm of databases. We note that certain key examples necessary for convergence may never be included in the training set as either positive or negative examples, and this leads to a potential deadlock situation. We label such instances with a new label value. In order to avoid deadlock, we are forced to add a new convergence rule to the existing version spaces algorithm.

We formally show how the modified algorithm correctly eliminates duplicate hypothesis candidates and candidates inconsistent with labeled examples in the training set. The modified algorithm preserves the original properties of Mitchell's Version Spaces. If the target query can be found in the search space, the algorithm will eventually converge on a single representative instance of the correct equivalence class, avoiding deadlock. If there are no candidates in the search space consistent with the training set, the algorithm will return an empty result.

Sample selection issues specific to the database problems were addressed by constructing queries on the database that either allow the system to construct an example with some useful features or to fix a series of missing labels to the same instances. We are able to produce a quantifiable bias for each feature, by exploiting catalog statistics to

derive an information gain measure. We argue that this bias corresponds to a significant imperative for the specific problem of database integration.

The result is a learning interface, initialized by an example constructed by the user with a simple drag and drop interface, and which keeps interacting with the user by fabricating more examples until it is satisfied that it has converged to the proper definition the user intended.

Chapter 6 Experimental Results

We implemented a prototype system complete with graphic-user interface. This prototype handles data mapping instances presented here, as well as mappings from meta-data elements to data [LSS96]. This small higher-order generalization allows from a broader range of restructuring queries across schematically disparate sources, without any substantial changes to the overall system.

We chose four domains to experiment with data integration. All of those problems were actual internet database integration tasks conducted under contract, in an ad-hoc fashion by a web services consulting firm. The schema is intact but the data have been altered or substituted when original data was not available. In some cases source data was available dynamically in HTML form, with the source sites wrapped ([BFG01], [CMM01]) to produce structured results. These experiments with Sphinx recreate those schema integration tasks, and are ranked in Table 2, by increasing level of empirical complexity. In the first domain the target queries populate a Healthcare local provider directory database. The second domain is based on local sport league statistics databases. The third domain is the '5 Star Catalog' for electronics and comes from the area of online pricing catalogs for B2B merchandise distributors. A slightly simplified version is used as an illustrative example in this exposition. The last domain involves data migration between two application platforms for Gene Expression Microarray data management.

The '5 Star Catalog' is the example closest to the prototype database in classic SQL textbooks. Domain specific considerations and context are examined in detail in previous chapters, and the schema mapping is illustrated in Figure 18. We will provide similarly detailed exposition of the other three data integration problems here. Experimental validation of the Sphinx bias on a set of canonical applications is not possible since no such benchmarks exist in the heterogeneous database integration literature [FLM98]. Rather than focus our analysis on a schema integration problem of

our own design, like our scientific predecessors we sought anecdotal, real world data integration case studies. The resulting analysis will show that the bias in Sphinx is at least plausible and can successfully address real world problems.

1. Experimental Databases

Table 1 shows a quantitative profile of the 8 queries selected for experimenting with Sphinx. The following section details the real applications from which those queries were chosen.

1.1. Rio Grande Valley Healthcare Providers

The various tables of an in-house database require normalization in order to be published on-line. The work is in two parts. The first task is to reorganize the data so as to make it accessible from a single source and a single table. This is necessary in order to interface the data with the e-commerce suite that will drive the web site. The second task is to build the web site displays and allow user queries. Only the first of these two tasks is of interest here.

This domain different tables, each of which fit into one of three categories: PeopleSUBCAT, PeopleCAT, PlacesCAT. Tables in the first two categories share the same schema. Tables in the third category have a different schema. The following two categories share the same schema:

PeopleSUBCAT = {Physicians, Dentists, Mental Health}

PeopleCAT = {Podiatrists, Optometrists}

The tables in both contain 13 columns:

(lname, fname, minitial, title, suffix, address, city, state, zip, phone, specialty, date_created, date_modified)

All the tables in the third category share the same schema:

PlacesCAT = {Medical Equipment, Nutrition and Wellness, Home Health, Pharmacies, Rehabilitation}

The tables in *PlacesCAT* contain 12 columns:

(name, name2, address, city, state, zip, phone, category, subcategory, date_created, modifed, name_exp)

- Federating Query

This domain requires all the data to be integrated into a single federated view, from which standard web querying tools can operate a web site. A few trivial domain mismatches are present in the data, such as the spelling of `Physicians' in the schema, versus 'Physician' in the data, or 'Rehabilitation' versus 'Senior Living Options'. Provided such can be resolved the following SchemaSQL federating query will define a federated view.

CREATE VIEW 'Directory' (name1, name2, middle, suffix, title, address, city, state, zip, phone, category, subcategory, date_created, date_modified, name_expanded)

AS (SELECT r.lname, r.fname, r.minitial, r.suffix, r.title, r.address, r.city, r.state, r.zip, r.phone, T, r.specialty, r.date_created, r.date_modified

FROM PeopleSUBCAT T

T r

*WHERE *)*

UNION

(SELECT r.lname, r.fname, r.minitial, r.suffix, r.title, r.address, r.city, r.state, r.zip, r.phone, T, null, r.date_created, r.date_modified

FROM PeopleCAT T

T r

*WHERE *)*

UNION

(SELECT r.name, r.name2, null, null, null, r.address, r.city, r.state, r.zip, r.phone, T, r.category, r.date_created, r.modified, r.name_exp

FROM PlacesCAT T

T r

*WHERE *);*

- Test Query 1

SELECT r.lname, r.fname, r.minitial, r.suffix, r.title, r.address, r.city, r.state, r.zip, r.phone, T, r.specialty, r.date_created, r.date_modified

FROM People_Table T

T r

*WHERE *;*

- Test Query 2

SELECT r.name, r.name2, null, null, null, r.address, r.city, r.state, r.zip, r.phone, T, r.category, r.date_created, r.modified, r.name_exp

FROM Places_Table T

T r

*WHERE *;*

1.2. Sport Statistics Databases

A local league, keeps a web site that regroups various information, including statistics about the players. The information is contributed semi-officially by each team manager and kept on files on separate computers. As result, the league database is a collection of each team's database files, rather than a coherently designed single entity. The goal of the project was to make all this information available under one database for the web site designers.

- Team Database

Each franchise or team has a database kept locally by a team statistician or a parent. This database has a table that contains a list of the players. The schema for that table is the following:

(Num, Name, Position, Age, Ht, Wt, Born, Birthdate).

- League Database

Each player has an extensive set of individual stats. There is one table per player. The table contains that player's career stats. The schema for each player stat table is different depending on whether the player is a goaltender or a position player. The name is the player is also the name of the table for that player's stats, unless two players share the same name. The schema for position players is:

(YR, TM, GP, G, A, PTS, PIM, +/-, PPG, SHG, SHOTS, PCT)

The schema for goaltenders is:

(YR, TM, GP, W, L, T, MIN, GA, SO, GAA, SPCT)

- Federated Views

There are three federated views for this domain data:

- The first view contains the list of all the position players and all goaltenders. Its schema is:

  (Name, Position, Team, Shoots, Height, Weight, Birth Date, Place of Birth, Current residence, Year Drafted, Round Drafted, Overall Choice, Number, Current Status, Compensation, Biography)

- The second view contains all the statistical data for all position players. Its schema is:

  (Season, Team/League, GP, G, A, TP, PIM, +/-, PP, SH, GW, GT, Shots, Pct)

- The third view contains all the statistical data for all goaltenders. Its schema is:

  (Season, Team/League, GP, W, L, T, MIN, GA, SO, AVG, PIM, Shots, Pct, A)

- Federating Queries

Each of those views can be populated from the input databases with SchemaSQL federating queries.

CREATE VIEW `Players' (Name, Position, Team, Shoots, Height, Weight, Birth Date, Place of Birth, Current residence, Year Drafted, Round Drafted, Overall Choice, Number, Current Status, Compensation, Biography)

AS SELECT y.Name, y.Position, T, null, y.height, y.Weight, y.Birthdate, y.Born, null, null, null, null, y.Num, null, null, null

FROM Teams T

T y

*WHERE *;*

CREATE VIEW 'Position-Stats' (Season, Team/LEAGUE, GP, G, A, TP, PIM, +/-, PP, SH, GW, GT, Shots, Pct

AS SELECT y.YR, y.TM, y.GP, y.G, y.A, y.PTS, y.PIM, y.+/-, y.PPG, y.SHG, null, null, y.SHOTS, y.PCT

FROM Players P

P y

Teams T

T w

WHERE w.Name = P AND

(w.Position = 'Left Wing' OR w.Position = 'Right Wing' OR w.Position = 'Center' OR w.Position = 'Defense');

CREATE VIEW 'Goaltender-Stats' (Season, Team/League, GP, W, L, T, MIN, GA, SO, AVG, PIM, Shots, Pct, A)

AS SELECT y.YR, y.TM, y.GP, y.W, y.L, y.T, y.MIN, y.GA, y.SO, y.GAA, null, y.SPCT, null

FROM Players P

P y

Teams T

T w

WHERE w.NAME = P AND w.Position='Goaltender';

- Test Query 3

SELECT y.Name, y.Position, T, null, y.height, y.Weight, y.Birthdate, y.Born, null, null, null, null, y.Num, null, null, null

FROM Teams T

T y

*WHERE *;*

- Test Query 4

SELECT y.YR, y.TM, y.GP, y.G, y.A, y.PTS, y.PIM, y.+/-, y.PPG, y.SHG, null, null, y.SHOTS, y.PCT

FROM Players P

P y

Teams T

T w

WHERE w.Name = P AND w.Position = "position";

1.3. 5 Star Catalog

This is the domain used as a running example throughout. It is inspired from an e-commerce workspace for Liaison's Content Exchange rich information extraction software. There are three tables: Product Catalog, Product Reviews and User Reviews. Their schema are the following:

Product Catalog (SKU, Manufacturer, Name, Description, Price, Orderinformation)

Product Reviews (Product_ID, Product_Type, Manufacturer, Name, EditiorialReview, Rating, SalesRank, Memory, DisplayType, DisplayColors, DisplaySize, DisplayResolution, OS, PCCompatible, MACCompatible, Serial, USBSupport, Software, Width, Height, Depth, Weight, WarrantyPart, WarrantyLabor)

User Reviews (Product_ID, Comment, Rating)

- Test Query 5

SELECT pc.Manufacturer, pc.Name, pc.Description,
pr.EditorialReview, pr.Rating, pr.SalesRank, ur.Comment,
ur.Rating, pc.Price

FROM ProductCatalog pc,

ProductReview pr,

UserReview ur

WHERE pc.Name=pr.Name AND
pc.Manufacturer=pr.Manufactuerer

AND pr.Product_ID = ur.Product_ID;

- Test Query 6

SELECT pc.Manufacturer, pc.Name, pc.Description,
pr.EditorialReview, pr.Rating, pr.SalesRank, ur.Comment,
ur.Rating, pc.Price

FROM ProductCatalog pc,

ProductReview pr,

UserReview ur

WHERE pr.Rating='5-star'

AND pc.Name=pr.Name AND pc.Manufacturer=pr.Manufactuerer

AND pr.Product_ID = ur.Product_ID;

1.4. SMD → BASE Data Migration

The Stanford Microarray Database (SMD) is one of the early packages used to drive Gene Expression Microarray production systems. As a result, the standard schema used by SMD is still common: the more advanced LAD system uses the same schema at its core [KSI03].

BASE is an open source system [STV02], built from the ground up as a competing platform for Microarray data management. Because of its open source nature, many new analysis tools have become available for BASE as users of the system can contribute new tools. BASE uses a different schema, but both SMD and BASE store closely related data. Although data processing steps vary between the two systems, the raw data formats are fundamentally compatible. The goal of our application is to migrate data from the database of the older system (SMD) to the database of the newer system (BASE), so that users can avail themselves of BASE analysis and export tools without any modification to their SMD platform.

- SMD Schema

  The 3 tables of interest for SMD data are:

Result	**Experiment**	**Expt_desc**
exptID	exptID	exptID
spot	printid	description
suid	organism	date_created
printid	category	created_by
date_modified	subcategory	date_modified
ch1i_mean	experimenter	modified_by
ch1d_median	exptname	
ch1i_median	slidename	
ch1_per_sat	gridfile	
ch1i_sd	ch1file	
ch1b_median	ch2file	
ch1b_sd	ch1desc	
ch1d_mean	ch2desc	
ch2i_mean	expttype_no	
ch2_per_sat	ref_id	
ch2i_sd	software	
ch2b_mean	scanversion	
ch2b_median	is_reverse	
ch2b_sd	gif_dir	
ch2bn_median	expt_date	
ch2in_median	date_created	
corr	created_by	
diameter	date_modified	
flag	modified_by	

log_rat2n_mean	scanparam	
log_rat2n_median	remarks	
pix_rat2_mean		
pix_rat2_median		
pergtbch1i_1sd		
pergtbch1i_2sd		
pergtbch2i_1sd		
pergtbch2i_2sd		
rat1_mean		
rat1n_mean		
rat2_mean		
rat2_median		
rat2_sd		
rat2n_mean		
rat2n_median		
regr		
sum_mean		
sum_median		
tot_bpix		
tot_spix		
x_coord		
y_coord		
top		
bot		
left		
right		

- BASE Schema

  The two tables of interest are:

RawBioAssayData	**RawBioAssay**
rawBioAssay	Id
position	name
element	descr
molecule	user
block	addedDate
numCol	imageAquisition
numRow	labeledCh1
x	labeledCh2
y	featureDate
dia	featureSoftware
FCh1Median	filename

FCh1Mean	useCount
FCh1SD	spots
BCh1Median	
BCh1Mean	
BCh1SD	
percCh1SD1	
percCh1SD2	
percCh1Sat	
FCh2Median	
FCh2Mean	
FCh2SD	
BCh2Median	
BCh2Mean	
BCh2SD	
percCh2SD1	
percCh2SD2	
percCh2Sat	
ratiosSD	
rgnRatio	
rgnR2	
Fpixels	
Bpixels	
flags	
Mvalue	
CV	

- Federating and Test Queries
    - Test Query 7

INSERT INTO rawbioassaydata

("rawBioAssay", "position", "FCh2Mean", "FCh2Median", "percCh2Sat", "FCh2SD",

"BCh2Mean", "BCh2Median", "BCh2SD", "FCh1Mean", "FCh1Median", "percCh1Sat",

"FCh1SD", "BCh1Mean", "BCh1Median", "BCh1SD", "dia", "flags",

"percCh2SD1", "percCh2SD2", "percCh1SD1", "percCh1SD2",

"ratiosSD", "rgnRatio", "BPixels", "FPixels", "x", "y",

"MValue", "CV", "reporter", "block", "numCol", "numRow")

SELECT '100',

r.spot,

r.ch1i_mean,

r.ch1i_median,

r.ch1_per_sat,

r.ch1i_sd,

r.ch1b_mean,

r.ch1b_median,

r.ch1b_sd,

r.ch2i_mean,

r.ch2i_median,

r.ch2_per_sat,

r.ch2i_sd,

r.ch2b_mean,

r.ch2b_median,

r.ch2b_sd,

r.diameter,

r.flag,

r.pergtbch1i_1sd,

r.pergtbch1i_2sd,

r.pergtbch2i_1sd,

r.pergtbch2i_2sd,

r.rat2_sd,

r.regr,

r.tot_bpix,

r.tot_spix,

r.x_coord,

r.y_coord,

0,

0,

1,

1,

1,

1

FROM result r;

- Test Query 8

INSERT INTO rawbioassay

(id, name, descr, owner, filename, spots, "addedDate", "imageAcquisition",

"worldAccess", removed)

SELECT '100',

ex.exptname,

exd.description,

'2',

ex.gridfile,

(select count(*) from result r where r.exptID=ex.exptID)[1]*,*

ex.date_created,

'1',

'1',

'0'

FROM experiment ex, expt_desc exd

WHERE ex.exptid = exd.exptid;

2. Experimental Setup and Results

We run a set of three experiments. The first experiment establishes the performance of Sphinx in learning test queries 1 through 8 and measures the impact of exploiting predicate information gain bias derived from catalog statistics. The second experiment compares the performance of the learning algorithm with Sphinx choosing examples versus an oracle choosing examples. The oracle draws on knowledge of the available example base, the target query and Version Spaces. The oracle picks examples that minimize the number of steps necessary to converge to the result. The third experiment compares the performance of Sphinx choosing examples with the performance of an unsupervised passive learner choosing examples at random.

2.1. Baseline vs. Catalog Statistics Strategy

The basic approach of Sphinx is to search for positive examples both to accelerate the exploration of the search space. We propose two basic strategies for sample selection: S_b the baseline strategy, and S_c a strategy incorporating catalog statistics. Both strategies start by building an identical search space. Both strategies look for examples that negate a

[1] this aggregation function is part of the correct view definition but it is optional and is replaced by a '0' in our experiments

small fraction of all potential features. Potential features are ranked by order of likelihood, first join predicates, then selection predicates from most likely to least likely. Strategy S_c selects the 10 least likely and tries to negate them. Baseline strategy S_b, also selects the bottom 10 predicates, however the ranking is based on a random draw, rather than catalog statistics. The ranking method statistical versus random is the sole difference between the strategies. Both heuristics incorporate bias based on Occam's Razor, betting that none of the negated predicates appear in the target query.

The number of features (i.e. Selection and Join predicates) appearing in the *Where* clause of each query is shown (e.g. 1J, 0S: 1 Join and no Selection predicates) and is roughly equal to the number of attributes in the tables used to define the target view. Table 2 shows the number of examples Sphinx requires to reach the target query using heuristic Sc and using heuristic Sb. Depending on the initial example chosen by the user to build the search space, results will vary. Data points for each query are collected by averaging over ten runs using different initial examples. The first column of Table 2 shows the number of features (i.e. Selection and Join predicates) appearing in the *Where* clause of each query. The second column shows the number of potential features for each query. Subsequent columns show for each strategy, positive, negative and total number of examples to converge to the result.

Strategy S_b is more successful than S_c for the target queries which do not have a selection predicate in their *Where* clause. The performance of strategy S_b degrades quickly when the target query contains even a single selection predicate. Strategy S_c, shows a more stable behavior, its performance only slowly decreasing when the complexity of the target query increases. This can be attributed to the inability of strategy S_b to differentiate among the selection features, and hence pick the ones, which can be excluded from the target query with high probability.

2.2. Sample Selection Heuristic vs. Optimal

The next experiment is to compare the rate of convergence when examples are chosen by the sample selection heuristics of Sphinx, against the rate of convergence when

an oracle (us) selects examples. The oracle has total knowledge, of the target query, of the learning algorithm and of the databases. The oracle (us) selects the best examples in the data, to allow the learning algorithm to converge in as few steps as possible. The result obtained by the oracle is optimal and represents an absolute lower bound.

Table 3 compares the performance of Sphinx when examples are selected with heuristics S_c and S_b vs. when the oracle selects examples. The oracle knows the exact target query. Sphinx does not, but its heuristics afford a reasonable bias allowing it to perform within the same order of magnitude.

2.3. Sphinx vs. Unsupervised Random Learner

The Clio system proposes an unsupervised learning process: the user chooses new examples to submit to the system until he or she is satisfied that "the transformations that result are what she intended" [YMH01]. We modified Sphinx to function as a passive learner in which the user carries the burden of constructing data mapping examples and labeling them. Sphinx merely indicates when the system has converged to a target query. We simulate a naïve user choosing examples at random from the set DM, such that at each step there is an equal probability of a positive or a negative example being chosen. Each example is picked randomly from its respective population of positive (DM^+) or negative examples (DM^-) with uniform probability distribution over the space of example feature vectors. Unlike the oracle this user is not in a feedback loop with the learning system, and does not know which examples are redundant with previously picked examples.

Table 3 compares the performance of Sphinx example selection mechanism to this random selection strategy. We observe that unsupervised random example selection performs an order of magnitude worse than Sphinx whenever the target query has at least one predicate in it. In effect, the premise of unsupervised learning is problematic. If a learner does not select examples, as Sphinx does, it is up to the user to become an expert and pilot the system. We note that Clio introduces the notion of refinement operators, allowing the user to manipulate a new algebra in order to guide Clio to learn the correct

query [YMH01]. Sphinx also takes that burden away from the user, in addition to deciding when the search has converged to a result.

3. Conclusion

A weakness of experimental database integration system is the paucity of application testing. Experimental applications are harder to come by than with typical general-purpose query interfaces. Typically each database integration effort involves its own engineering challenge, and federated database systems are often identified with a canonical application. This phenomenon tends to limit the scope of testing and the conclusions that can be drawn.

We did not encounter nor were we able to locate real-world federating queries containing more than a few straightforward natural joins. We did not seek to contrive such queries, although some examples can be found in the literature. Doing so would neither prove nor disprove the bias underlying Sphinx since such a bias must be observed empirically in order to be validated. As to whether Sphinx can learn any arbitrary, general-purpose query, the answer to that question is clear: the futility of bias-free learning imposes itself when one considers the exponential size of the search space and our criteria for a provably correct solution, rather than a best estimate. This last premise was dictated by the interface concept itself. Should one abandon it and turn Sphinx into a best guess tool, the existing sample selection bias can be easily converted into a more traditional candidate ranking.

As we demonstrated the Sphinx prototype with an application, we are drawn to some conclusions. The number of examples necessary for convergence does not pose a problem despite the size of the search spaces, and we observe that our bias is definitely applicable. Further as the size of databases grows, the statistical classification becomes more significant as the separation between enumerated and infinite types become more pronounced. These experiments show that even with imperfect heuristics, the observed complexity is correlated with the size and complexity of the target query rather than with

the number of potential features. We observe that the number of required examples spikes when predicates are added to the target query. However the number of required examples grows slowly even when the number of potential features is large (showing an exponential growth in the search space). We also demonstrate the benefits of supervised learning with Sphinx over an unsupervised approach relying on the user to choose examples.

				Strategy Sc			Strategy Sb		
		Query size	Potential Features	Number of Examples			Number of Examples		
				Total	Pos.	Neg.	Total	Pos.	Neg.
Healthcare	Query 1	0J, 0S	14	2.2	2.2	0.0	1.4	1.4	0.0
	Query 2	0J, 1S	15	4.7	2.0	2.7	5.5	2.5	3.0
Sports Statistics	Query 3	1J, 0S	25	4.9	3.7	1.2	2.5	1.5	1.0
	Query 4	1J, 1S	25	4.8	1.8	3.0	11.4	1.6	9.8
5 Star Catalog	Query 5	2J, 0S	30	5.9	3.2	2.7	4.0	2.0	2.0
	Query 6	2J, 1S	30	8.9	2.7	6.2	11.7	2.0	9.7
SMD → Base	Query 7	0J, 0S	57	4.5	4.5	0.0	x	x	x
	Query 8	1J, 0S	35	3.2	2.2	1.0	3.0	2.0	1.0

Table 2 – Number of examples required to converge. Two heuristics compete.

		S_c	S_b	Oracle			Random		
		Total	Total	Total	Pos.	Neg.	Total	Pos.	Neg.
Healthcare	Query 1	2.2	1.4	1	1	0	26	13	13
	Query 2	4.7	5.5	2	1	1	26	13	13
Sports Statistics	Query 3	4.9	2.5	2	1	1	7.5	3.5	3.0
	Query 4	4.8	11.4	3	1	2	22	11	11
5 Star Catalog	Query 5	5.9	4.0	4	1	3	12	5.5	6.0
	Query 6	8.9	11.7	5	1	4	42	21	21

Table 3 – Number of examples required to converge. Sphinx selects examples vs. an oracle vs. random selection

Chapter 7 Extending Sphinx to XML Data Integration

By definition semi-structured data is less regular than relational data. As a consequence algebra and calculus for semi-structured data use simplified models and present more complexity than their relational counterpart. The groundwork necessary for extending Sphinx to XML is concerned with identifying a formal framework for data integrating transformations. This initial effort requires dealing with a myriad of possibilities with no clear structuring principle. Fernandez, Siméon and Wadler defined the W3C approved algebra for XML ([FSW01]). This algebra uses a simplified model of XML data and data types combined with an iterative for/loop constructor. To simplify, whereas relational databases and their algebra are loosely based on sets and set operators, XML requires the use of forests in lieu of sets, element operators in lieu of set operators, and a loop iterator over forests of elements.

1. XML Algebra

Without loss of generality, we adopt the simplified schema convention for XML from the semi-monad algebra of Fernandez et al., which is the official W3C algebra. An XML schema definition can be summarized as a list of element definitions. Each element is defined as either an atomic element, or a complex element. A complex element definition consists of a name and a list of atomic or complex elements, optionally annotated with the quantification attributes *, + or ?. An atomic element definition consists of a name and a one of three simple types: Integer, Boolean and String.

Since XML schema is modeled as a tree with labeled node, we will make heavy use of the partial order relation “ancestor or equal to”. We will denote LCA(S) the least common ancestor for a set of nodes S in an XML schema. MSC(S) will represent the minimum spanning clade for a set of nodes S: i.e. the whole subtree rooted at LCA(S).

2. Restructuring and schema mappings.

Before approaching the wider problem of multiple schema integration we look at the more elementary problem of transforming XML data documents to reflect a simple schema update. Thus we will consider the data to be stored in a document and consider what it means to move this document from its current XML schema to a different XML schema. We will not yet consider the problem of data integration: that is while the data is restructured we will not consider joins/selections or any interaction with potentially other relevant data. Yet, we believe that resolving this initial step of schema mapping will yield insight into the more complex task. Further, even if that were not the case, this simple restructuring problem constitutes a subset of the greater integration problem, which must necessarily be resolved.

We abstract the problem by defining the notion of a *schema mapping function* between a source and a destination schema. The schema mapping function specifies for each point in the destination a mapping to the origin in the source.

Definition: *Schema Mapping Function*

A *schema mapping function* φ between a destination XML schema Xd and a source XML schema Xs, is defined for each atomic element x in Xd, by $\varphi(x)$ is either null or an atomic element of Xs. φ is a schema mapping function if for $\varphi(x)=\varphi(x')$ implies $x=x'$. That is φ is an injective mapping from Xd into Xs.

2.1. Closure

We define the notion of closure for schema mapping functions, with respect to the partial order.

Definition: *Closure (of Schema Mapping Functions)*

Given a schema mapping function φ from XML schema Xd into Xs, φ° is the *closure* of φ, defined by:

If $\varphi(x) = \text{null}$, $\varphi^{\circ}(x) = \text{null}$

If $\varphi(x) \neq \text{null}$, $\varphi^{\circ}(x) = \text{LCA} (\cup_{\forall y \in MSC(x)}\{\varphi(y)\})$

$$= LCA(\varphi(MSC(x)))$$

Note that φ° is no longer an injective mapping from Xd to Xs. But there are two noteworthy properties of φ°:

Closure Properties

(P1). Assume $\varphi(x) \neq null$, if x is an ancestor or equal to y then $\varphi^{\circ}(x)$ is an ancestor or equal to $\varphi^{\circ}(y)$

(P2). Assume $\varphi(x) \neq null$, $\varphi^{\circ}(x) = LCA(\varphi^{\circ}(MSC(x))$

We adopt the convention for the above definitions and formulae that null is the empty set $\varnothing$, that LCA ($\varnothing$) = $\varnothing$, and that any element is the ancestor of null. The proof for P1 is left to the reader, and P2 can be proven by applying P1.

Proof for (P1):

For x = y, (P1) is trivial.

Assume $\varphi(x) \neq null$, $\varphi(y) = null$, then $\varphi^{\circ}(x)$ is an ancestor or equal to $\varphi^{\circ}(y)$.

Assume $\varphi(x) \neq null$, $\varphi(y) \neq null$ and x is an ancestor of y.

Because x is an ancestor of y, $MSC(y) \subseteq MSC(x)$ and therefore $\varphi(MSC(y)) \subseteq \varphi(MSC(x))$.

Since LCA is monotonic: $\varphi(MSC(y)) \subseteq \varphi(MSC(x))$ implies $LCA(\varphi(MSC(y)))$ is an ancestor or equal of $LCA(\varphi(MSC(x)))$.

Therefore by definition of φ°: $\varphi^{\circ}(x)$ is an ancestor or equal of $\varphi^{\circ}(y)$.

■

Proof for (P2):

For any y in MSC(x), x is ancestor of y, and using (P1), $\varphi^{\circ}(x)$ is an ancestor of $\varphi^{\circ}(y)$.

So $\varphi^{\circ}(x)$ is ancestor of $\varphi^{\circ}(MSC(x))$ and $\varphi^{\circ}(x)$ is ancestor of $LCA(\varphi^{\circ}(MSC(x)))$

$\varphi(MSC(x)) \subseteq \varphi^{\circ}(MSC(x)$ by definition of φ°: $\varphi(x)=y$ implies that $\varphi^{\circ}(x)=y$. Since LCA is a monotonic function: $X \subseteq Y$ implies LCA(Y) is an ancestor or equal to LCA(X).

Therefore $LCA(\varphi^{\circ}(MSC(x))$ is an ancestor or equal to $LCA(\varphi(MSC(x)) = \varphi^{\circ}(x)$.

By anti-symmetry: $\varphi^{\circ}(x) = LCA(\varphi^{\circ}(MSC(x)))$.

■

2.2. Top-Down Completion

In order to complete the process started by defining a closure for φ, we want to define completion for a schema mapping, in order to yield a full relation: i.e. a function where every element has a non-null image. This requires giving a mapping value to some nodes in order to replace the null value. We define completion by top-down inheritance: each node if it is mapped to a null value will instead be mapped to the same node as its direct ancestor or parent.

Definition: *Top-Down Completion (of Schema Mapping Functions)*

Given a schema mapping function φ from XML schema Xd into Xs, we define φ^1 the *top-down completion* of φ as:

$$\varphi^1(Root(Xd)) = Root(Xs)$$

$$\varphi^1(x) = \varphi(x), \text{ if } \varphi(x) \neq null$$

$$= \varphi^1(parent(x)), \text{ if } \varphi(x) = null.$$

Properties of Top-Down Completion:

(P3). Given a schema mapping function φ from XML schema Xd into Xs,

such that φ(root(Xd)) = root(Xs) and φ obeys properties P1 and P2,

then the completion of φ, φ^1 obeys properties P1 and P2.

2.3. Invariant Schema Restructuring of an XML document.

In this section we will define a method which enables us to migrate a single document D with XML schema Xs into a restructured document D' with XML schema Xd. We baptize this method invariant restructuring because as much of the nesting associations of the original document D are conserved in D' as the new schema Xd will allow. Thus we conjecture that our restructuring represents the minimal information loss for D by measure of nesting associations.

We consider an initial mapping consisting of three parts:

- A source XML schema Xs
- A destination XML schema Xd
- A mapping function φ, such that for each atomic element in x in Xd, φ(x) is either null or an atomic element of Xs.

We assume that neither source, nor destination schema are circular, and that quantification of element definition is limited to three values: *none*, +, and *, indicating that the element type is either a single node, or a *forest* (*collection* in the traditional XML terminology) of 1 to many nodes, or a forest of 0 to many nodes. We assume that φ is such that for any node x, the cardinality of x is compatible with the cardinality of φ (x), i.e.: *none* is compatible with *none*, + is compatible with +, and * is compatible with *, + and *none*.

Further, without loss of generality, we assume that the root of Xd is mapped to the root of Xs. If this is not the case, it is sufficient to prune Xs to MSC(φ(Xd)) and map Root(Xd) to Root(MSC(φ (Xd))).

Using the XML semi-monad algebra, we define document D' the restructuring of document D by the mapping φ. φ^1 is derived from φ by applying closure and completion.

```
D' =    root(Xd) [ for n in children(root(Xd)) do convert_node(n, D) ]
type UrLeaf = ~[UrScalar]
type UrInterior = ~[UrTree+]
```

```
type UrTree = UrLeaf | UrInterior

fun convert_node(n : UrTree; source_context : UrTree) : UrTree =
        match n
                case leaf : UrLeaf do
                                                        #n is a leaf node in Xd
                        if φ^1(leaf) = source_context then
                          leaf [ D/data() ]
                        else
                                for x in D/φ^1(leaf) do
                                                #x is an instance of D mapped to position n
                                where x in source_context/φ^1(leaf) do
                                        leaf [ x/data() ]
                case interior : UrInterior do
                                                                #n is an interior node in Xd
                        if φ^1(interior) = source_context then
                                interior [ for m in children(interior) do
                                convert_node(m, source_context)]
                        else
                                for x in D/φ^1(interior) do
                                        where x in source_context/φ^1(interior) do
                                                interior [ for m in children(interior) do
                                                        convert_node(m, x) ]
                else error()
```

We can also construct the XML query ρ(φ) by syntactic recursion over Xd. ρ(φ) will convert D into D'. We assume schema Xs and Xd are themselves expressed in the form of XML documents in XML schema, each node $n of schema Xs and schema Xd has two attributes: $n.xpath(), $n.label(). These attributes return respectively: the XPath specification, and the XML schema label associated with a node n.

```
D':             Let     $n = Xd
                                        #initializes $n as the root of document Xd
                        $r = D
                                          #initializes $r as the root of document D
                Return
                        <$n.label()>
                        $r.data()
                        For $m in [$n/*/]
                                             #iterates over the children of $n in Xd
                        Return
```

```
                    F($m, $r)
                </$n.label()>

function F($n, $source_context):
    For $x in [D/φ^1($n).xpath()]
    Where contains($source_context, $x)
    Retur
        <$n.label()>
        $x.data()
            For $m in [$n/*/]
                                        #embedded query, which
                                #iterates over the children of $n in Xd
                                    #(noop if $n is a leaf node)
            Return
                F($m, $x)
        </$n.label()>
```

Example:

Consider the schema definition for Xs and Xd

```
type Xs =   r[      s [     a [String]*
                            t [     b [String]*
                                    u[      c [String]*]*
                              ]*
                    ]*
            ]
type Xd =   R [     S[      A [String]*
                            B [String]*
                            C [String]*
                    ]*
              ]
and φ: A → a, B → b, C → c, R → r.
D = <r>     <s>     <a>a01</a><a>a02</a>
                    <t>     <b>b01</b>
                            <u><c>c01</c><c>c02</c></u>
                    </t>
                    <t>     <b>b02</b>
                            <u><c>c03</c><c>c04</c></u>
                    </t>
            </s>
            <s>     <a>a03</a>
                    <t>     <b>b03</b>
                            <u><c>c05</c></u>
```

```
        </t>
    </s></r>
```

By completion and closure we have $\varphi^1(S)=s$. And we can build D':

```
D' = <R>    <S>   <A>a01</A><A>a02</A>
                  <B>b01</B><B>b02</B>
                  <C>c01</C><C>c02</C><C>c03</C><C>c04</C>
              </S>
              <S>   <A>a03</A>
                  <B>b03</B>
                  <C>c05</C >
              </S></R>
```

Note that the grouping in D of {a01, a02, b01, b02, c01, c02, c03, c04} and {a03, b03, c05} in separate sub-trees has been preserved in D'. However there is some information loss since the groupings {b01, c01, c02}, {b02, c03, c04} and {b03, c05} have not been preserved.

2.4. Restructuring with Joins

Two source schema Xs_1, and Xs_2 can, without loss of generality be merged into one new schema Xs, by introducing a new node root(Xs), with two repeatable children elements: root(Xs_1) and root(Xs_2). Thus we can without consider the federation of several documents into one, as we would the restructuring of a single document. However to be effective our approach must deal with joins, and conserve inherent join dependencies properties, in the same fashion that it dealt with conserving nesting dependency properties.

Assume that Q and R are two nodes in Xs, such that Q has a child element QID of cardinality 1, which joins with RID, a child element of R with cardinality 1. We modify the tree-shaped schema Xs, by adding two new directed arrow: one which makes QID a parent of R, and another which makes RID a parent of Q. We define a new element QR corresponding to that join, such that the children of Q and the children of R are also children of QR. We can replace both Q and R by QR is Xs. This leads to the duplication

of QR and its children element in Xs. This does not pose a problem for the computation of the MSC function. As for the computation of LCA, the least common ancestor, since there are duplicate elements, there are several possibilities depending on the choice of permutations. By choosing the most specific result (w.r.t. the ancestor or equal partial ordering) for LCA: we conserve an important property of LCA, which is to find the shortest path between two points. In the source document D itself, each *collection* (*forest* in the terminology of [FSW01]) of Q and R node instances will be replaced by a QR collection formed, by joining each instance in the original collection with its counterpart(s).

All properties of closure and completion are conserved, regardless of Xs no longer being a proper tree graph. These adaptations to our method allow us to apply our restructuring algorithm in the presence of a join between two collections Q and R.

Note: *An unexplored possibility could be to turn Xs into a directed graph with cycles, by adding an edge from Q to R, and an edge from R to Q. Modification of D to reflect the new edges in Xs would be fairly straightforward: each Q and R instance node would acquire as children the set of nodes with which it joins. However algorithms for LCA and MSC must still be adapted.*

Two questions remain: is a join between Q and R what the user desires, and, is the nesting and join dependencies conserved by our restructuring yield the correct nesting properties desired by the user. To answer the first question requires building a search space of potential joins and use examples to determine the desired output. The second issue is more complex because the search space for all possible nesting combinations is exponential, however we will seek to address that problem by proposing to restructure the data according to a number of possible normal forms.

3. Nesting and Normal Forms

We have sought a restructuring method, which would preserve as much information as the new schema allows on nesting relationships between elements in each

collection. However in certain cases either because not enough information can be preserved, or because a new nesting scheme logically imposes itself to the user, we must allow the specification of new nesting dependencies between element collections. Rather than consider arbitrary re-arrangement of the document tree, we introduce a set of potential normal forms and associated operators, by which manipulation of the document tree will be possible. Note that XML schema contains no nesting information: it is the instantiation of the schema into a document tree that nesting alternatives are created.

Example:

```
D1 = <R>    <S>   <A>a01</A>
                  <T>   <B>b01</B>
                  </T>
                  <T>   <B>b02</B>
                  </T>
            </S></R>

D2 = <R>    <S>   <A>a01</A>
                  <T>   <B>b01</B>
                  </T>
            </S>
            <S>   <A>a01</A>
                  <T>   <B>b02</B>
                  </T>
            </S></R>
```

D_1 and D_2 are two alternate instantiations of the same schema. They contain the same *set* of atomic elements, and differ by the nesting relationships between elements.

3.1. Nested Relational Data in PNF

XML allow us to consider the storage and manipulation of semi-structured data in a fairly general sense. However, it is not customary for XML data available today to be arbitrarily semi-structured. For practical considerations, most XML documents are designed rationally in an ER modeling sense and conform to an XML schema. Most data served as XML originates from relational or nested-relational databases due to strong commercial imperatives. This further increases the regularity of XML documents.

As a result it can be reasonable to model XML Schema using the nested relational model [YMH01]. Despite XML Schema's significant expressive power, the nested relational model provides us with enough tools to describe the core features of XML: atomic nodes and repeatable complex nodes.

3.2. Partitioned Normal Form for XML

We borrow the concept of partitioned normal form from nested relational algebra, and we will extend it to cover XML documents.

Definition: *Nested Relational Partitioned Normal Form*

A nested-relational table is in *partitioned normal form*, PNF if and only if, for each row the atomic elements form a key.

Property:

(P4) If T is in partitioned normal form, ∀e non-atomic element in table T

$$Unnest_e(Nest_e(T))=T$$

We define an equivalent concept for XML: an XML document in *partitioned normal form* is such that for each element, the atomic members of that element form a key: no instances of the same element share the same atomic values.

Definition: *Atomic or Leaf Node*

We define an *atomic node*, or a *leaf node* in XML as an element that cannot have elements as its children. Given an element, the function get_atomic_children returns the sub-collection of non-atomic elements.

```
type UrTree = UrLeaf | UrInterior
type UrInterior = ~[UrTree+]
type UrLeaf = ~[UrScalar]

fun is_atomic (c: UrTree) : Boolean =
        match c
          case leaf: UrLeaf do true
          case interior: UrInterior do false
```

```
        else error()
fun get_atomic_children (c: UrInterior) : UrLeaf* =
      for x in children(c) do
        where is_atomic(x) do x
```

By extension we will also refer to a node N in a schema Xd for document D, as a leaf node or an atomic node if the corresponding element instances of N in D are atomic (i.e. elements of D/N are atomic).

Definition: *XML Partitioned Normal Form*

A document D is in partitioned normal form if and only if the function is_PNF(D) returns true.

```
fun is_PNF (n : UrTree) : Boolean =
      match n
            case leaf: UrLeaf do
                  true
            case interior: UrInterior do
                  if empty( for x in children(interior) do
                                                #test if no children share same
                        for y in children(interior) do#atomic elements
                              where get_atomic_children(x) =
                              get_atomic_children(y) do
                                    x
                        for x in children(interior) do
                                                #test if all non atomic children
                              if is_PNF(x) then()
                                                      #are also in PNF
                              else x)
            else error()
```

Example:

Document D_1 above is in PNF, but document D_2 is not. If we take the syntactic liberty of applying the nested relation operators $Nest_T$ and $Unnest_T$ to D_1, and D_2, we notice D_1 is invariant and that D_2 transforms into D_1, which is in PNF.

$$Nest_T(Unnest_T(D_1)) = D_1$$

$$Nest_T(Unnest_T(D_2)) = D_1$$

Successive applications of the nest and un-nest operators for all interior nodes of a document, is one way to format an arbitrary document in PNF. Because XML data is stored on relational databases, most XML data, if modeled as nested relational, is found in *partitioned normal form*. This is because such XML document, are produced by successive applications of the nest operator on a flat universal relation. Since a non-nested universal relation is in *partitioned normal form* by default, thanks to property (P4), nest and un-nest operators become left and right inverse of each other and partitioned normal form is conserved for any sequence of applications of nest and un-nest operators.

3.3. Production Normal Form

We define *production normal form* as a relaxed case of *partitioned normal form*. Production normal form has the characteristic of being the only format, which can be produced by restructuring of partitioned normal form documents, using only pure loop iterators, without any conditional loop iterators: i.e. *for* loops without a *where* clause.

In the example document D_3, the atomic element of S, which is A, forms a key for S. But in example document D_4, two instances of S with the same value of a01 appear, first in the same instance as c01, then in the same instance as c02. In D_4, the cardinality of S is dictated by the cardinality of its sub-element U: there is only one instance of U embedded in each instance of S.

Thus we will distinguish two kinds of repeatable sub-elements: the first dictates the cardinality of their parent element, by requiring a repeat instance of the parent element for each instance of the sub-element. This is the same behavior as a non-repeatable element and we will refer to those as expanded sub-elements. The second type of sub-element appears in more condensed form: two sub-elements whose parents are atomically equal will always appear simultaneously, rather than in separate instances of their parent. This second kind of sub-element we will refer to as collapsed. In the example D_4, U is a production element, whereas T is a collapsed element.

Example:

```
D3 = <R>    <S>   <A>a01</A>
                  <T>   <B>b01</B>
                  </T>
                  <T>   <B>b02</B>
                  </T>
                  <U>   <C>c01</C>
                  </U>
                  <U>   <C>c02</C>
                  </U>
            </S></R>

D4 = <R>    <S>   <A>a01</A>
                  <T>   <B>b01</B>
                  </T>
                  <T>   <B>b02</B>
                  </T>
                  <U>   <C>c01</C>
                  </U>
            </S>
            <S>
                  <A>a01</A>
                  <T>   <B>b01</B>
                  </T>
                  <T>   <B>b02</B>
                  </T>
                  <U>   <C>c02</C>
                  </U>
            </S></R>
```

Definition: *Collapsed Elements*

Given a document D, with XML schema Xd, a node N in Xd, with a non-root parent node P; Elements instances of N are *collapsed* in D if and only if one of the following properties is true for any pair (p, p') of element instances of P in D (i.e. elements of the collection D/P):

1. p and p' are equal

2. if all atomic children elements of p and p' are equal and p and p' have an identical parent element, then p and p' have identical collections of children elements with tag N.

The following is_collapsed function provides an algebraic test for elements N with a parent element P, and a grandparent element G.

```
type UrTree = UrLeaf | UrInterior
type UrInterior = ~[UrTree+]
type UrLeaf = ~[UrScalar]

fun is_collapsed (D: UrTree): Boolean =
        empty(for g in D/G do
                        for p1 in g/P do
                                for p2 in g/P do
                                        if get_atomic_children(p1) =
                                        get_atomic_children(p2) then
                                                if p1/N = p2/N then ()
                                                else g
                                        else ())
```

Note: *The above definition of collapsed requires the equality of collections for children elements with tag N. In the relational world the term set would be preferred, however in the XML world two elements may share the same set of elements in a different order, and still not be considered equal. Requiring equality of collections rather than sets, makes production normal form a more regular format. The equality symbol '=' in the algebraic definition of is_collapsed also refers to collection equality.*

Definition: *Duplicates*

Two elements x and x' are *duplicates* if and only if for any element tag T: the collection x/T is equal to x'/T.

Property:

(P5) A document D with schema Xd, such that for every node N with parent P in Xd, the instances of N in D are collapsed is in partitioned normal form if D has no pair of duplicate elements.

Proof:

Take D a document with no duplicate elements, with schema Xd, such that for every node N with parent P and grandparent G, in Xd, the instances of N in D are collapsed.

Assume Z is a non-root repeatable non-atomic node of Xd with parent G. Take two instances z and z' of Z in document D, such that all atomic elements of z and z' are equal, and z and z' have the same parent g.

Assume Y is a non-atomic child node of Z in Xd: by definition of D, all instances of Y in Z are collapsed, which means that collection z/Y is equal to collection z'/Y.

Thus, for any Y, child node of Z in Xd: z/Y = z'/Y. This is our definition for duplicate elements: z = z'.

Assume that Z is a root, atomic or non-repeatable node of Xd: Z has at most one child therefore any atomic elements that child may have form a key.

It can easily be verified that D is in partitioned normal form since we have established that for any node Z, its atomic elements form a key.

Definition: *Expanded Elements*

Given a document D, with XML schema Xd, a non-root node N in Xd, with parent node P; Elements instances of N are *expanded* in D if and only if for any element instance p of P in D (i.e. an element of the collection D/P): p has at most one element instance of P (p/N is a single element collection or empty)

Definition: *Production Normal Form*

A document is in *production normal form*, PxNF if and only if any repeatable non-root element with a repeatable parent element is either collapsed, expanded or both.

Partitioned normal form is a special case of production normal form, in which all non-root elements are collapsed.

3.4. Production Normal Form Representation

We define the PNF operator which takes document D and output D', which is in Partitioned Normal Form. We define the operator x_A: which takes a document D, and produces document D', where element A is in expanded form.

Definition: *Expansion Operator*

Let Xd be a schema Xd, such that N is a non-root node in Xd, with parent P. The operator *expansion operator* x_N takes a document D and returns D', such that instances of N in D' are expanded.

```
fun x_N(D: UrTree) : UrTree =
    match D
        case leaf: UrLeaf do
            leaf
        case interior: UrInterior do
            if empty(interior/N) then
                x_N(interior)
            else for c in children(interior) do
                where name(c)=N do
                    name(interior) [ (for k in children(interior) do
                                        where name(k)≠N^2 do k)
                                    c]
        else error()
```

The reader will observe that given a document D in PxNF with schema Xd, and any two nodes A and B in Xd, operators x_A and x_B are commutative and idem-potent: $x_A(x_A(D)) = x_A(D)$, $x_A(x_B(D))=x_B(x_A(D))$. Further PxNF is *closed* with respect to any expansion operator x_N: if D is in PxNF, then $x_N(D)$ is in PxNF.

Definition: *make_PNF Operator*

We define the operator *make_PNF*, which takes any document and returns a document in PNF.

---

[2] We are taking the liberty of introducing the negation operator, which is syntactic sugar for reversing the consequence clauses in the if-then-else construct.

```
fun make_PNF (D: UrTree): UrTree =
        match D
          case leaf: UrLeaf do
                distinct(children(leaf))
          case interior: UrInterior do
                if empty(get_atomic_children(children(interior))) then
                                                        #interior has no atomic child
                  name(interior) [
                        for x in distinct(children(interior)) do
                                make_PNF(x)
                  ]
                else
                        let keylist = distinct(for c in children(interior) do
                        keys[get_atomic_children(c)])
                                        #keylist= list of key values for children of interior
                                for x in keylist do
                                                #for each key value make a new element
                                  name(interior) [ x/keys/data()
                                                        #put in the atomic elements first
                                    let match_key = for k in children(interior) do
                                      where x/keys/data()=get_atomic_children(k) do
                                        k
                                    for y in match_key do
                                        for z in children(y) do
                                          if is_atomic(z) then ()
                                          else make_PNF(x)
                                            #add the non-atomic children of the children
                                                        #of interior which match the key
                                  ]
          else error()
```

The reader will observe that the make_PNF operator is idem-potent, although it is not commutative with any of the expansion operators.

Given a document D with schema Xd, we can apply the PNF operator to D, yielding D'=make_PNF(D). Then we can apply a set of expansion operators x_N (where interior node N in schema Xd).

4. Combining Restructuring with Formatting Operators: Syntactic Conjecture

Given a document D with schema Xs in PNF, a single query can restructure D into D' a document in PxNF with schema Xd.

Consider the following schema Xs, schema Xd and mapping φ:

```
Xs =   Q [
           P [
               A[string]*
               B[string]*
           ]*
       ]
Xd =   R [
           S [
               T [
                   AA [string]*
               ]*
               BB [string]*
           ]*
       ]
φ:     AA -> A
       BB -> B
φ^1:   AA -> A
       BB -> B
       T -> A
       S -> P
       R -> Q
```

The XML query ρ(φ) will convert D into D'.

```
ρ(φ) = Let    $d = D
       Return
           <R>
               For $s in [D/P]
               Where contains($d, $s)
               Return
                   <S>
                       For $t in [D/A]
```

```
                                    Where contains($s, $t)
                                    Return
                                        <T>
                                            For $aa in [D/A]
                                            Where contains($t, $aa)
                                            Return
                                                <AA>
                                                $aa.data()
                                                </AA>
                                        </T>
                                    For $bb in [D/B]
                                    Where contains($s, $b)
                                    Return
                                        <B>
                                        $bb.data()
                                        </B>
                            </S>
                </R>
```

Since logically it is always the case that $aa = $t, consider simplifying ρ(φ) by removing the redundant variable.

```
ρ(φ) = Let      $d = D
                Return
                        <R>
                                For $s in [D/P]
                                Where contains($d, $s)
                                Return
                                        <S>
                                                For $t in [D/A]
                                                Where contains($s, $t)
                                                Return
                                                        <T>
                                                                <AA>
                                                                $t.data()
                                                                </AA>
                                                        </T>
                                                For $bb in [D/B]
                                                Where contains($s, $b)
                                                Return
                                                        <B>
                                                        $bb.data()
                                                        </B>
```

```
                    </S>
            </R>
```

Consider the following document D:

```
D = <P>
        <Q>
            <A>a01</A>
            <A>a02</A>
            <B>b01</B>
            <B>b02</B>
        </Q>
```

We can restructure D into D’ = ρ(φ)(D).

```
D’ = <R>
        <S>
            <T>
                <A>a01</A>
            </T>
            <T>
                <A>a02</A>
            </T>
            <B>b01</B>
            <B>b02</B>
        </S>
    </R>
```

It would make sense, in order to preserve the original structure to collapse A, yielding $co_A(D')$, where co_A is the collapse operator for node A.

```
coA(D’) = <R>
        <S>
            <T>
                <A>a01</A>
                <A>a02</A>
            </T>
            <B>b01</B>
            <B>b02</B>
        </S>
    </R>
```

The query obtained by operator composition is $co_A \bullet \rho(\varphi)$. We note that $co_A \bullet \rho(\varphi)$ can also be derived from $\rho(\varphi)$ by moving the For/loop block iterator in bold, deeper into the structure of the query by one increment.

$co_A \bullet \rho(\varphi)$ =

```
Let     $d = D
                Return
                        <R>
                                For $s in [D/P]
                                Where contains($d, $s)
                                Return
                                        <S>
                                                <T>
                                                        For $t in [D/A]
                                                        Where contains($s, $t)
                                                        Return
                                                                <AA>
                                                                $t.data()
                                                                </AA>
                                                </T>
                                                For $bb in [D/B]
                                                Where contains($s, $b)
                                                Return
                                                        <B>
                                                        $bb.data()
                                                        </B>
                                        </S>
                        </R>
```

Now consider expanding B and collapsing A in D', this yields $x_B \bullet co_A$ (D').

$x_B \bullet co_A$ (D') =

```
        <R>
                <S>
                        <T>
                                <A>a01</A>
                                <A>a02</A>
                        </T>
                        <B>b01</B>
                </S>
                <S>
                        <T>
```

```
                    <A>a01</A>
                    <A>a02</A>
                </T>
                <B>b02</B>
            </S>
        </R>
```

The composed query is $x_B \bullet co_A \bullet \rho(\varphi)$. We note that it is derived from $co_A \bullet \rho(\varphi)$ by un-nesting the For/loop block in bold by one increment.

```
xB•coA•ρ(φ) =
Let     $d = D
            Return
                <R>
                    For $s in [D/P]
                    Where contains($d, $s)
                    Return
                        For $bb in [D/B]
                        Where contains($s, $b)
                        Return
                            <S>
                                <T>
                                    For $t in [D/A]
                                    Where contains($s, $t)
                                    Return
                                        <AA>
                                        $t.data()
                                        </AA>
                                </T>
                                <B>
                                $bb.data()
                                </B>
                            </S>
                </R>
```

To generalize this special case, we conjecture that any arbitrary series of collapse and expand operators applied upon a document in PNF can be expressed in XML algebra within the syntactic framework of a fixed template where the placement of for/loop block iterators determines the sequence of applicable operators

Syntactic Conjecture. Given a mapping φ between Xs and Xd, the set of queries formed by the conjugation of ρ(φ) with expansion and collapse operators for any arbitrary sets of nodes in Xd is a syntactic space, where any element can be constructed from ρ(φ) by nesting and un-nesting of the for/loop block iterators within the tree-shaped structure of the output template

5. Conclusion

This syntactic conjecture if verified, represents the possibility of building a search space of XML transformations easy to represent and to implement. Collapse and expand operators form a group acting upon the set of XML documents in production normal form that are derived from ρ(φ). The conjecture defines a homomorphism from that group to a language subset of XML algebra sentences constructed from a fixed template. Thus, the conjecture allows us to define a search space that is both syntactically and algebraically interesting.

- Since expand and collapse operators are idem-potent and commutative (expands commutes with expand, collapse commutes with collapse, although expand only commutes with collapse when restricted to document in production normal form) our search space can be represented by a finite feature vector.
- The syntactic conjecture defines for each feature vector representation, a unique syntactic query derived from a single template, with a pre-set order in which elements must appear.
- This syntactic representation gives a natural way of incorporating selection predicates into our search space. Selection predicates, once defined, can be incorporated as a further set of filters composed with the queries in the search space by syntactic insertion in the Where position of available for/loop block iterators.

Chapter 8 Limitations and Conclusion

1. Limitations of our Approach

To place this work into a larger context we review first what is currently within the scope of our candidate elimination active learning model, and second of our implemented feasibility prototype, Sphinx. We speak to the scope of features and predicates that can be addressed in this model and those that must be addressed in other ways.

1.1. Scope of the Work

As our goals were to push the envelope on a fully automated system, we did not handle unit conversion and data dictionary issues. Unit conversions can be ounces to grams, months to quarters, and other sorts of data misrepresentations. The construction of whole synonym dictionaries or thesauri is characterized by its own set of technical challenges ([PHC95], [TMH95], [DDH01]) and is well outside the scope of our study.

Furthermore incorporating key generating (or skolem constant generating) functions would allow the vertical partitioning of source data into separate target views. This has not yet been addressed although we believe this would not pose any theoretical difficulty within our framework.

1.2. Limitations in the Construction of the Search Space

Our feasibility prototype incorporates a fully automated way of building a search space with a limited class of queries (equality predicates) and with no user intervention beyond the initial drag and drop construction. It was not possible to invent arbitrary predicates such as negation or comparison, because of the infinite number of possibilities

consistent with the initial data mapping example give by the user. Specifying arbitrary predicates for inclusion in the search space of potential features, would require a user to provide additional input. Our goal would be to allow the user to do so without requiring the user to learn SchemaSQL syntax, perhaps in the same way that spreadsheets (such as MS Excel) allow specification of basic algebra without requiring programming skills.

We did not consider a number of theoretical issues such as inclusion of the aggregation and sorting operators. Adding such functionality to a basic schema integration queries, would also require more input provided by more advanced users.

1.3. Limitations of Adopting the Version Spaces Model

The Version Spaces model can handle any query expression formed with a conjunction of arbitrary predicates. Predicates may contain any arithmetic such as 'x.a+3>y.a', or more complex features such as negation 'x.a≠y.a', outer joins '(x.a=y.a or x.a=null or y.a=null)', disjunction, etc.. Such queries are expressible with the algebraic formula shown in Figure 17b and fall within the expressive power of Version Spaces and of the active learning algorithm for which we elaborated formal correctness and termination guarantees. However, even though such predicates may contain disjunctions, when we build the search space we seek to form a new view definition by combining potential predicates on a purely conjunctive basis.

Another limitation of Version Spaces is linked to the completeness of the considered search space. When a target query is in that search space, given a sufficient training set, Version Spaces will converge to a query defining the correct target view. If Version Spaces returns an empty result, the target query is not in the search space. And if the target query is not in the search space, it is possible for the algorithm to converge to a single query consistent with labeled examples in the training set, when inclusion of additional examples in the training set would drive Version Spaces to an empty result.

1.4. Limitations of the Implemented Prototype

In generating a default search space, we explicitly curtailed Sphinx to a limited class of queries with equality predicates. Without additional input from the user, currently generated potential predicates include all legal equality selection predicates on objects involved in the initial data mapping. They also include all legal equality join predicates on those same objects with the exception of self-joins and of a peculiar kind of join, characterized by join paths in the query graph that traverse children of data mapping elements. As shown by the Clio system, a more exhaustive, though still not complete, generation of all equality selection and join predicates is possible, by building a query graph and taking into account every possible join path between relations used in the attribute mapping. It is also possible to generate a number of potential outer and inner joins predicates besides the natural joins.

2. Machine vs. User Responsibilities

In making simplifying assumptions in this approach we tried to distinguish several taxonomy elements. This allows us to take advantage of the taxonomy, and divide responsibilities between system and user such that the systems will learn those elements that are difficult to specify by a non-technical user, and we leave the user to handle elements that are not easily learnable by an automated system.

One noteworthy is the semantically complex specification of relational operator such as the join. This element is difficult for the user to specify, and requires abstract knowledge of relational concepts. On the other hand, it is extremely easy, via examples, for a system to establish whether a join is involved or not.

Another noteworthy is the thesaurus synonyms and unit conversion problem. It is not easy for a machine learning system to establish with certainty that ‘”P III” and “Pentium 3” are synonyms. We note that it is completely trivial for even the most novice user to specify this kind of conversion. This might however become tedious and

repetitive and as such, automating this problem represents a research challenge of its own.

3. Open Problems

Besides the catalog and unit conversion issues, which are the focus of other work in the literature, three limitations of Sphinx remain unresolved challenges.

We did not address the issue of specifying arbitrary predicates, aggregation, group-by and sort operators as part of the view defining queries. Those operators and predicates are neither straightforward for a user to specify, nor easy for a machine to deduce based on examples. Spreadsheets provide such operations for their user with a semi-graphical interface, and could arguably serve as a model to integrate these operations ex-post.

The issue of integrating some form of disjunction remains open. Whether this happens at the predicate level or at the sentence level, providing a limited form of disjunction would be useful. We believe that specifying the aggregation (horizontal union) of multiple learned queries into a single target view would not pose a challenge. However providing some form of disjunction at the predicate level is an unresolved issue. Disjunction could either become part of the user input choices, or could be integrated into the learning algorithm or both. As expressed in our taxonomy, integrating unrestricted disjunction at the sentence level without some additional kind of user input is futile: we observed that a bias-free learner would merely seek a disjunction of all positive examples. Nor is it clear whether examination of real world applications would reveal this to be a useful feature in practice.

A different kind of challenge is to look beyond the relational model, and consider XML as a data model and XQuery as a data manipulation language. XML is a more general model and we expect its resulting taxonomy is more diverse than relational. As a consequence the range of tools provided by user interfaces and the range of expertise required of users is already far greater than those for relational data. However as XQuery and its algebra ([FSW01]) are far more imperative than their relational equivalent, the

problem of elaborating an example-based, programming by demonstration type interface for XML remains open.

4. Summary

We looked at the problem of heterogeneous database integration. We note that the main obstacle is the complexity and breadth of the task requiring challenging work by database specialists. We seek a solution which tackles the two fundamentally difficult aspects of database specification: reasoning about meta-data with a higher order language, and providing an intelligent interface allowing users to specify higher order queries in a more congenial environment. We introduced an efficient compilation mechanism as the basis for an execution engine for higher order view definitions in SchemaSQL. We built an interface centered on an active learning system, which reads a user given example. It elaborates further examples in an interaction that concludes in the complete specification of the higher order view definition sought by the user. We adopt the version spaces approach, preferring to trade extra examples and learning speed in exchange for accuracy. We propose a new tri-labeled version spaces algorithm, which bridges the gap between existing version spaces and a sample selection method. We exploit database catalog statistics in order to provide the bias necessary to overcome the exponential component of version spaces. Consistent with our approach, rather than use this bias to rank solution candidates, we seek faster convergence to a demonstrably correct result.

We demonstrate the Sphinx prototype on an application. Despite its current limitations outlined above, we feel that by addressing the core of the learning issues associated with disambiguating the input from a point and click interface, Sphinx validates the concept of interactive specification. Adding more features such as outer joins and simple comparison predicates or range predicates could be addressed within the same framework with almost no additional user input. Addressing other issues such as aggregation, arbitrary predicates and the relational union operator would probably require new forms of graphic input and interaction such as spreadsheet-inspired manipulations.

With this GUI prototype, we explore how far a fully automated system with almost no user interaction can go. This is an area that has traditionally challenged our ability to propose GUIs suited for non-technical users. Results highlight both the success as well as the limitations of our approach and ultimately prove our original concept of interactive and intelligent heterogeneous database integration.

References

[ACM97] Abiteboul S., S. Cluet, T. Milo (1997): Correspondence and Translation for Heterogeneous Data. ICDT 1997: 351-363

[AK93] Arens, Knoblock. “SIMS: Retrieving and Integrating Information From Multiple Sources”. Proceedings of the ACM SIGMOD Conference 1993

[AKS96] Arens Y., C. Knoblock, WM. Shen (1996): Query Reformulation for Dynamic Information Integration. JIIS 6(2/3): 99-130.

[ALS96] Andrews A., N. Shiri, L. V.S. Lakshmanan, I. N. Subramanian. “On Implementing SchemaLog – A Database Programming Language”, Proceedings CIKM Conference (1996): 309-316.

[BM75] Bledsoe W.W., J.T. Minor: “Unskolemizing”. University of Texas at Austin, Math Department, technical report ATP-77.

[BM01] Barbançon F., D. Miranker. Compiling Higher-Order Federating Queries for Execution on Relational Systems. University of Texas at Austin, dept. of Computer Sciences TR01-41. http://www.cs.utexas.edu/users/francois/tr01-41.pdf

[BM04] Barbançon F., D. Miranker (2004): Active Learning of Schema Integration Queries. The University of Texas at Austin, Dept. of Computer Sciences Tech. Report CS-TR-04-23 (submitted for publication).

[BFG01] Baumgartner R., S. Flesca, G. Gottlob (2001): Visual Web Information Extraction with Lixto. VLDB 2001: 119-128.

[CDA99] Castano S., De Antonelli V. (1999): A schema analysis and reconciliation tool environment. IDEAS 1999: 53-62.

[Chamberlin76] Chamberlin D.D. (1976): Sequel 2: a unified approach to data definition, manipulation and control. IBM Journal of Research and Development 20(6):560-575.

[CKW93] W. Chen, M. Kifer, D. Warren: “HiLog, A Foundation for Higher-Order Logic Programming”. J. Logic Programming 1993:15:187-230.

[CB97] Chen Y., W. Benn: Building DD to Support Query Processing in Federated Systems. KRDB 1997: 5.1-5.10.
[CDS98] Sophie Cluet, Claude Delobel, Jêrome Siméon, Katarzyna Smaga: “Your Mediators Need Data Conversion!” SIGMOD Conference 1998: 177-188.
[CAL94] Cohn D., L. Atlas, R. Ladner: Improving Generalization with Active Learning. Machine Learning 15(2): 201-221 (1994).
[CH94]W. W. Cohen, H. Hirsh: The Learnability of Description Logics with Equality Constraints. Machine Learning 17(2-3): 169-199 (1994)
[CMM01] Crescenzi V., G. Mecca, P. Merialdo: RoadRunner (2001): Towards Automatic Data Extraction from Large Web Sites. VLDB 2001: 109-118.
[Codd70] Codd E.F. (1970): A relational model of data for large shared data banks. Communications of the ACM 13(6):377-387.
[CAL92] Cohn D., L. Atlas and R. Ladner: Improving Generalization with Active Learning. Machien Learning 15(2): 201-221 (1994)
[DE95] Dagan I., S. Engelson: Committee-Based Sampling for Training Probabilistic Classifiers. ICML 1995: 150-157.
[DDH01] Doan A., P. Domingos, A. Halevy (2001): Reconciling Schemas of Disparate Data Sources: A Machine Learning Approach. SIGMOD Conference 2001.
[DLD04] Dhamankar D., Y. Lee, A. Doan, A. Y. Halevy and P. Domingos: iMAP: Discovering Complex Mappings between Database Schemas. SIGMOD Conference 2004: 383-394
[FMS01] Fernandez M., A. Morishima, D. Suciu (2001): Efficient Evaluation of XML Middle-ware Queries. SIGMOD Conference 2001.
[FSW01] Mary Fernandez, Jérôme Siméon, and Philip Wadler (2001): “A semi-monad for semi-structured data”. ICDT 2001: 263-300.
[FLM98] Daniela Florescu, Alon Y. Levy, Alberto O. Mendelzon: ”Database Techniques for the World-Wide Web: A Survey”. SIGMOD Record 27(3): 59-74 (1998)
[GS95] Gasarch W., C. Smith: Recursion Theoric Models of Learning: Some Results and Intuitions. Annals of Mathematics and Artificial Intelligence, 15:151-166 (1995).

[GMP97] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, Jennifer Widom: “The TSIMMIS Approach to Mediation: Data Models and Languages”. JIIS 8(2): 117-132 (1997).

[Haussler88] D. Haussler: Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework. Artif. Intell. 36(2): 177-221 (1988).

[Hirsh91] Hirsh H.: Theoretical Underpinnings of Version Spaces. IJCAI 1991, 665-670.

[Hirsh92] Hirsh H.: Polynomial-Time Learning with Version Spaces. AAAI 1992: 117-122.

[HKW97] Laura M. Haas, Donald Kossmann, Edward L.Wimmers, Jun Yang: “Optimizing Queries Across Diverse Data Sources”. VLDB 1997: 276-285

[HMP97] Hirsh H., N. Mishra, L. Pitt: Version Spaces without Boundary Sets. AAAI/IAAI 1997: 491-496.

[Kent91] Kent W. (1991): Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language. VLDB 1991: 147-160

[Kent92] Kent W. (1992): Profile Functions and Bag Theory. Hewlett-Packard Company. Technology Department, Hewlett-Packard Laboratories. Palo Alto, California.

[KGK95] W. Kelley, S. Gala, W. Kim, T. Reyes, B. Graham: “Schema Architecture of the UniSQL/M Multidatabase System”. Modern Database Systems 1995: 621-648

[KSI03] Killion P.J., G. Sherlock, V.R. Iyer: The Longhorn Array Database (LAD): An Open-Source, MIAME compliant implementation of the Stanford Microarray Database (SMD). BMC Bioinformatics 2003, **4**:32 (2003).

[KLK91] Krishnamurthy R., W. Litwin, W. Kent (1991): Language Features for Interoperability of Databases with Schematic Discrepancies. SIGMOD Conference 1991: 40-49.

[KQ03] Koriche F., J. Quinqueton: Robust *k*-DNF Learning via Inductive Belief Merging. ECML 2003: 229-240.

[Kuhns67] Kuhns J.L. (1967): Answering questions by computer: a logical study. Rand Corp.

Technical Report RM-5428-PR.
[SW94] K. Sagonas, D. Warren: “A Portable Compiler for Integrating HiLog into Prolog Systems”. SLP 1994: 682.
[LSS96] Lakshmanan, Sadri, Subramanian. “SchemaSQL – A Language for Interoperability in Relational Multi-Database Systems”, appeared in the proceedings of VLDB 1996 Conference in Bombay.
[LSS97] Lakshmanan, Sadri, Subramanian. “Logic and Algebraic Languages for Interoperability in Multidatabase Systems”. Appeared in Journal of Logic Programming, November 1997.
[LSS99] Lakshmanan, Sadri, Subramanian. “On Efficiently Implementing SchemaSQL on an SQL Database System”. VLDB 1999: 471-482.
[LRO96] Levy A., A. Rajaraman, J. Ordille (1996): Querying Heterogeneous Information Sources Using Source Descriptions. VLDB 1996: 251-262.
[]Lewis D., J. Catlett: Heterogeneous Uncertainty Sampling for Supervised Learning. ICML 1994:148-156.
[LCS00] Li W., C. Clifton, S. Liu (2000): SemInt: a tool for identifying attribute correspondences in heterogeneous databases using neural network. Data and Knowledge Engineering 33(1): 49-84.
[]Liere R., P. Tadepalli: Active Learning with Committees for Text Categorization. AAAI 1997: 591-596.
[Lifschitz96] V. Lifschitz: "Foundations of logic programming," in Principles of Knowledge Representation, CSLI Publications, 1996, pp. 69-127
[Lloyd87] John Lloyd. “Foundations of Logic Programming”. Springer Verlag, 1987. Second, extended edition.
[MBR01] Madhavan J., P. Bernstein, E. Rahm (2001): Generic Schema Matching with Cupid. VLDB 2001: 49-58.
[Michalski83] Michalski R.S. : A Theory and Methodology of Inductive Learning. Artif. Intell. 20(2): 111-161 (1983).

[Miller98] R. Miller: “Using Schematically Heterogeneous Structures”. Proceeedings of the ACM SIGMOD Conference 1998:189-200

[MHH00] Miller R., L. Haas, M. Hernández (2000): Schema Mapping as Query Discovery. VLDB 2000: 77-88.

[MF90] Muggleton S., C.Feng: Efficient Induction of Logic Programs. ALT 1990: 368-381.

[Murray87] K.S. Murray: Multiple Convergence: An Experiment in Disjunctive Concept Acquisition. The University of Texas at Austin, Dept. of Computer Sciences Tech. Report CS-TR-AI-87-56.

[MZ98] Milo T., S. Zohar (1998): Using Schema Matching to Simplify Heterogeneous Data Translation. VLDB 1998: 122-133.

[MTB02] Miranker D., M. Taylor, A. Padmanaban: A Tractable Query Cache by Approximation. SARA 2002: 140-151

[Mitchell77] Mitchell T.: Version Spaces (1977): A Candidate Elimination Approach to Rule Learning. IJCAI 1977: 305-310

[Mitchell82] Mitchell T.: Generalization as Search. Artificial Intelligence, 18:203-226, 1982.

[MWK00] Mitra P., G. Wiederhold, M. Kersten (2000): A Graph-Oriented Model for Articulation of Ontology Interdependencies. EDBT 2000: 86-100

[NH93] Norton S., H. Hirsh:Learning DNF Via probabilistic Evidence Combination. ICML 1993:220-227.

[PTU00] Palopoli L., G. Terracina, D. Ursino (2000): The System DIKE: Towards the Semi-Automatic Synthesis of Cooperative Information Systems and Data Warehouses. ADBIS-DASFAA 2000: 108-117.

[PHC95] Park Y., Han Y., Choi K. (1995): Automatic Thesaurus Construction Using Bayesian Networks. CIKM 1995: 212-217.

[Quinlan90] Quinlan J. R.: Learning Logical Definitions from Relations. Machine Learning 5: 239-266 (1990).

[RB01] Rahm E., P. Bernstein (2001): A survey of approaches to automatic schema matching. VLDB Journal 10(4): 334-350.
[Ross94] K. Ross: “On Negation in HiLog”. J.Logic Programming 1994:18:1:27-53
[STV02] L.H. Saal, Troein C., Vallon-Christersson J., Gruvberger S., Borg A., Peterson C., BioArray Software Environment: A Platform for Comprehensive Management and Analysis of Microarray Data. Genome Biology 2002 **3**(8): software0003.1-0003.6.
[Sebag96] Sebag M.: Delaying the Choice of Bias: A Disjunctive Version Space Approach. ICML 1996: 444-452.
[Smirnov01] E.N. Smirnov: Conjunctive and disjunctive version spaces with instance-based boundary sets. PhD thesis, Dept. of Computer Science, Maastricht University, Maastricht, The Netherlands, 2001.
[SHS02] E. N. Smirnov, H. J. van den Herik, I. G. Sprinkhuizen-Kuyper: Adaptable Boundary Sets. AMAI 2002.
[SW95] K. Sagonas, D. Warren: “Efficient Execution of HiLog in WAM-based Prolog Implementations”. Proceedings on the twelth International Conference on Logic Programming, Tokyo, Japan 1995: 349-363.
[Sebag96] Sebag M.: “Delaying the Choice of Bias: A Disjunctive Version Space Approach”. ICML 1996:444-452.
[SPD92] Spaccapietra S., C. Parent, Y. Dupont: Model Independent Assertions for Integration of Heterogeneous Schemas. VLDB Journal 1(1): 81-126 (1992).
[Stephan95] Stephan F.: Learning via Queries and Oracles. COLT 1995:162-169.
[SWK76] Stonebraker M., E. Wong, P. Kreps and G. Held (1976): The design and implementation of Ingres. ACM Transactions on Database Systems 1(3):189-222.
[TMH95] Takenobu T., Makoto I., Hozumi T. (1995): Automatic Thesaurus Construction Based on Grammatical Relations. IJCAI 1995:1308-1313.
[TCM99] Thompson C., M. Califf, R. Mooney: Active Learning for Natural Language Parsing and Information Extraction. ICML 1999: 406-414.
[TRV96] Tomasic A., L. Raschid, P. Valduriez (1996): Scaling Heterogeneous Databases and the Design of Disco. ICDCS 1996: 449-457.

[Ullman82] Jeffrey D. Ullman: “Principles of Database Systems”, 2nd Edition. Computer Science Press 1982, ISBN 0-914894-36-6.

[VP97] V. Vassalos, Y. Papakonstantinou: “Describing and Using Query Capabilities of Heterogeneous Sources”. Proceedings of the 23rd VLDB Conference, Athens Greece, 1997.

[VRG98] M. E. Vidal, Louiqa Raschid, Jean-Robert Gruser: “A Meta-Wrapper for Scaling up to Multiple Autonomous Distributed Information Sources”. CoopIS 1998: 148-157

[Widom96] Jennifer Widom: Integrating Heterogeneous Databases: Lazy or Eager? ACM Comput. Surv. 28(4es): article 91 (1996).

[XQuery] XQuery: http://www.w3.org/TR/XQuery

[YOL97] Yan L., M. Özsu, L. Liu (1997): Accessing Heterogeneous Data Through Homogenization and Integration Mediators. CoopIS 1997: 130-139.

[YMH01] Yan L., R. Miller, L. Haas, R. Fagin (2001): Data Driven Understanding and Refinement of Schema Mappings. SIGMOD Conference 2001.

[Zloof77] Zloof M. (1977): Query-by-Example: A Data Base Language. IBM Systems Journal.

VITA

François Gérard Barbançon was born in Bangkok (Thailand) on April 4th 1973 to Gérard and An Barbançon. After graduating from the Lycée Kléber in 1990, he attended the Classes Préparatoires until 1992, when he entered the École Nationale Supérieure des Mines de Paris. After receiving the Diplôme d'Ingénieur Civil des Mines de Paris in 1995, he enrolled at the University of Texas at Austin in 1995, where he received an M.S. in Computer Sciences in 1998.

Permanent Address: PO BOX 7781,
AUSTIN TX 78713 USA
francois@cs.utexas.edu
http://www.cs.utexas.edu/users/francois

This dissertation was typed by the Author.